HAL
open science

Une méthode globale pour la vérification d'exigences temps réel : application à l'avionique modulaire intégrée

Michaël Lauer

To cite this version:

Michaël Lauer. Une méthode globale pour la vérification d'exigences temps réel : application à l'avionique modulaire intégrée. Réseaux et télécommunications [cs.NI]. Institut National Polytechnique de Toulouse - INPT, 2012. Français. NNT : 2012INPT0033 . tel-04264836v2

HAL Id: tel-04264836

https://theses.hal.science/tel-04264836v2

Submitted on 30 Oct 2023

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par :

Institut National Polytechnique de Toulouse

Spécialité :

Réseaux, Télécommunications, Systèmes et Architecture

Présentée et soutenue le 12 juin 2012 par :

Michaël Lauer

Une méthode globale pour la vérification d'exigences temps réel

Application à l'Avionique Modulaire Intégrée

Membres du jury

Frédéric Boniol	Professeur des universités	INP Toulouse - ONERA
Christian Fraboul	Professeur des universités	INP Toulouse - IRIT (président)
Serge Haddad	Professeur des universités	ENS Cachan - LSV
Frédéric Mallet	Maître de conférences	Université Nice Sophia Antipolis - INRIA (rapporteur)
Pascale Minet	Chargé de recherche	INRIA Le Chesnay (rapporteur)
Vlad Rusu	Chercheur	INRIA Villeneuve d'Ascq
Ye-Qiong Song	Professeur des universités	Université de Lorraine - LORIA

Ecole doctorale

Mathématiques, Informatique et Télécommunications de Toulouse

Unité de Recherche

Institut de Recherche en Informatique de Toulouse

Directeurs de thèse

Frédéric Boniol - Jérôme Ermont

Remerciements

Je tiens à remercier en premier lieu mon directeur de thèse, Frédéric Boniol, ainsi que mes encadrants, Jérôme Ermont et Claire Pagetti, sans qui je n'en serais certainement pas là. Merci pour leur patience, leur enthousiasme et leur confiance.

Je suis très reconnaissant envers Pascale Minet, chargée de recherche INRIA Le Chesnay, et Frédéric Mallet, maître de conférence à l'Université de Nice Sophia Antipolis et membre INRIA Sophia Antipolis Méditerranée, d'avoir accepté d'être les rapporteurs de cette thèse. Je remercie également Christian Fraboul, professeur des universités de l'Institut National Polytechnique (INP) de Toulouse et membre de l'Institut de Recherche en Informatique de Toulouse (IRIT), Serge Haddad, professeur des universités de l'Ecole Nationale Supérieure (ENS) de Cachan et membre du Laboratoire Spécification et Vérification (LSV), Vlad Rusu, chercheur INRIA Villeneuve d'Ascq, et Ye-Qiong Song, professeur des universités de l'Université de Lorraine et membre du Laboratoire lorrain de Recherche en Informatique et ses Applications (LORIA), d'avoir accepté de faire partie du jury. Je leur transmets toute ma gratitude pour leurs commentaires constructifs et les perspectives ouvertes lors des discussions qui ont suivi la soutenance de la thèse.

Durant cette thèse, j'ai eu l'occasion de travailler régulièrement dans les locaux de l'ONERA. Je remercie en particulier les membres du DTIM pour leur accueil, leur gentillesse et pour m'avoir invité à présenter mes travaux. Merci à Claire et Rémy de m'avoir trouver une place dans leur bureau.

Je remercie l'ensemble du personnel de l'IRIT, avec une attention particulière pour Sylvie Eichen et Sylvie Armengaud-Metche qui ont toujours été disponibles pour m'aider à gérer les formalités administratives. Encore merci à toutes les deux pour leur bonne humeur et leur efficacité. J'ai une pensé également pour tous les membres de l'équipe IRT et toutes les personnes avec qui j'ai partagé un bureau : Emmanuel, Julien, Gabriel, Tony, Adnan, Xiaoting et Nesrine. Cela a été un plaisir de travailler à leurs côtés.

Je ne peux terminer sans remercier de tout cœur ma famille, en particulier mes parents pour leur soutien et leurs encouragements ainsi que mes frères, Fabien et Olivier, qui m'ont inspiré cette aventure.

Et, enfin, je réserve mon ultime remerciement à toi, Adeline, mon épouse. Les figures de ce manuscrit ne seraient pas ce qu'elles sont sans ton œil avisé et ton sens critique. Mais par dessus tout, tu as compris et accepté mon choix de quitter un travail confortable pour me ré-orienter vers la recherche, ce qui m'a permis de m'y consacrer l'esprit libre. Je te dois tellement, merci d'avoir toujours été là.

Table des matières

Introduction **xiii**

Acronymes **xviii**

Notations **xxi**

I Contexte industriel et scientifique **1**

1 Avionique Modulaire Intégrée **5**
1.1 Architecture avionique modulaire intégrée 5
1.1.1 Module IMA 5
1.1.2 Réseau AFDX 6
1.2 Etude de cas : un système de gestion de vol 7
1.2.1 Fonctions 8
1.2.2 Architecture 9
1.2.3 Exigences 11
1.3 Conclusion 15

2 Méthodes de modélisation et de véri cation **17**
2.1 Modélisation d'architecture 18
2.1.1 Le langage de description d'architecture AADL 18
2.2 Modélisation de comportement 25
2.2.1 Approche avec automates temporisés et model checking 25
2.2.2 Le *tagged signal model* 30
2.2.3 Clock Constraint Specification Language 33
2.3 Evaluation de performances des réseaux embarqués 34
2.3.1 Network Calculus 35
2.3.2 Approche par trajectoire 39
2.4 Evaluation d'exigences de bout en bout 42
2.4.1 Evaluation de latence 43
2.4.2 Evaluation de fraîcheur 46
2.4.3 Evaluation de cohérence 47
2.5 Conclusion 48

3 Contribution **51**
3.1 Démarche suivie 51
3.1.1 Description de l'architecture 51
3.1.2 Formalisation des exigences et du comportement du système 52
3.1.3 Abstraction 52
3.1.4 Vérification 52
3.1.5 Publications 52
3.2 Développements réalisés 53
3.2.1 Un outil de vérification d'exigences temps réel 53
3.2.2 Intégration dans le projet SATRIMMAP 54

II Modèle et sémantique **57**

4 Modèle d architecture d un système IMA **61**
4.1 Architecture fonctionnelle . . . 61
4.2 Architecture matérielle . . . 64
4.3 Allocation sur la plateforme . . . 67
4.3.1 Allocation des fonctions . . . 68
4.3.2 Allocation des variables . . . 69
4.3.3 Allocation des dépendances . . . 71
4.4 Conclusion . . . 73

5 Modèle comportemental d un système IMA **75**
5.1 Définition des signaux utilisés . . . 75
5.1.1 Signaux pour les activations de fonction . . . 76
5.1.2 Signaux pour les copies de variable . . . 76
5.1.3 Domaine de définition des signaux . . . 76
5.2 Définition des processus utilisés . . . 77
5.2.1 Module . . . 77
5.2.2 Fonction . . . 77
5.2.3 Lisseur de trafic . . . 82
5.2.4 Port . . . 85
5.2.5 Capteur . . . 88
5.2.6 Actionneur . . . 89
5.2.7 Remote Data Concentrator . . . 90
5.3 Comportements du système complet . . . 91

6 Exigences **93**
6.1 Définitions préliminaires . . . 93
6.1.1 Chaîne fonctionnelle . . . 93
6.1.2 Dépendance étendue . . . 94
6.2 Exigences . . . 94
6.2.1 Latence . . . 95
6.2.2 Fraîcheur . . . 97
6.2.3 Cohérence . . . 98
6.2.4 Exigences de l'étude de cas . . . 102
6.3 Conclusion . . . 103

III Vérification des exigences **105**

7 Formalisation du problème **109**
7.1 Abstraction du problème . . . 109
7.1.1 Principes généraux . . . 109
7.1.2 Particularités du *tagged signal model* . . . 110
7.1.3 Abstraction des lisseurs de trafic . . . 111
7.1.4 Abstraction du réseau : canaux temporisés . . . 114
7.2 Intuitions pour la méthode de vérification . . . 118
7.3 Mise en œuvre et outillage . . . 120
7.3.1 Présentation générale . . . 121
7.3.2 Acquisition et création des objets . . . 121
7.3.3 Analyse réseau . . . 122
7.3.4 Constitution de l'ensemble des contraintes relatives à une exigence . . . 122

8 Résolution **123**
8.1 Encodage sous la forme d'un programme linéaire mixte . . . 123
8.1.1 Domaine de définition des variables . . . 123
8.1.2 Particularité pour la cohérence . . . 124
8.1.3 Représentation de l'infini . . . 126
8.1.4 Fonction objectif . . . 126
8.2 Fonctions objectifs par type d'exigence . . . 126
8.2.1 Exigence de latence . . . 126
8.2.2 Exigence de fraîcheur . . . 127
8.2.3 Exigence de cohérence entre chaînes fonctionnelles divergentes . . . 127
8.2.4 Exigence de cohérence entre chaînes fonctionnelles convergentes . . . 129
8.3 Application à l'étude de cas . . . 131
8.3.1 Exigence de latence . . . 131
8.3.2 Exigence de fraîcheur . . . 133
8.3.3 Exigence de cohérence entre chaînes fonctionnelles divergentes . . . 134
8.3.4 Exigence de cohérence entre chaînes fonctionnelles convergentes . . . 135
8.3.5 Conclusion . . . 135

9 Discussions **137**
9.1 Passage à l'échelle . . . 137
9.1.1 Expérimentations sur une étude de cas paramétrée . . . 137
9.1.2 Options du solveur . . . 138
9.1.3 Conclusion . . . 140
9.2 Ouverture vers des formules analytiques plus précises : application à la latence . . . 141
9.2.1 Un exemple simple . . . 141
9.2.2 Extension aux chaînes fonctionnelles emboîtées . . . 142

Conclusion et perspectives **147**

A Langage de description d architecture du prototype **151**

B Programmes linéaires des exigences de l étude de cas **157**

C Méthode analytique locale **161**

Bibliographie **177**

Table des gures

1.1 Exemple d'une MAF . . . 6
1.2 Fonctionnement d'un lisseur de trafic . . . 8
1.3 Chaînes fonctionnelles de l'étude du cas . . . 9
1.4 Architecture de l'étude de cas . . . 10
1.5 Ordonnancement des partitions du module 1 . . . 10
1.6 Latence d'une requête du pilote . . . 12
1.7 Fraîcheur de l'heure estimée d'arrivée . . . 13
1.8 Cohérence des affichages . . . 14
1.9 Cohérence des mesures de pression . . . 15

2.1 Composants définis dans AADL V1 . . . 18
2.2 Implémentation d'un *process* pour la fonction KCCU . . . 21
2.3 Implémentation du module M_1 . . . 23
2.4 Implémentation du système complet FMS . . . 24
2.5 Exemple d'automate temporisé . . . 26
2.6 Réseau d'automates temporisés du système FMS . . . 28
2.7 Automate temporisé de l'observateur . . . 28
2.8 Automate temporisé du clavier . . . 28
2.9 Automate temporisé modélisant la MAF du module M_1 . . . 29
2.10 Automate temporisé modélisant un canal temporisé . . . 29
2.11 Comportement d'un système avec P_{MFD_1} . . . 32
2.12 Comportement du système composé de P_{MFD_1} et P_{M_1} . . . 33
2.13 Courbes d'arrivée et de service de l'exemple . . . 35
2.14 Exemple d'une courbe d'arrivée . . . 36
2.15 Courbe d'arrivée du flux τ_i en entrée du réseau . . . 36
2.16 Courbe de service du nœud h . . . 37
2.17 Interprétation graphique des bornes de l'occupation d'un nœud et du délai . . . 38
2.18 Courbe d'arrivée d'un flux en sortie d'un nœud . . . 38
2.19 Principes de l'approche par trajectoire . . . 40
2.20 Fenêtre d'interférence de τ_j sur τ_i . . . 41
2.21 Paquets devant être comptabilisés deux fois dans $W_i^{last_i}(t)$ . . . 42
2.22 Un composant du *Real-Time Calculus* . . . 44
2.23 Courbes d'arrivée de *wpId* en entrée de FM_1 . . . 44
2.24 Courbes de service de FM_1 . . . 45
2.25 Pire latence d'une occurrence de *wpId* . . . 45
2.26 Latence de bout en bout . . . 46
2.27 Exemple de pessimisme de l'analyse de bout en bout avec le *Real-time Calculus* . . . 47
2.28 Exemple de système étudié . . . 48

3.1 Démarche suivie . . . 51
3.2 Principes de l'outil de vérification . . . 53
3.3 Processus du projet SATRIMMAP . . . 54

4.1 Dépendances des variables de la fonction *Flight Manager* . . . 62
4.2 Exemple de variables périodique (ETA_1) et sporadique ($wpInfo_1$) . . . 63
4.3 Exemple d'une variable sporadique dépendant de deux variables . . . 63

4.4 Graphe de dépendance des variables . . . 64
4.5 Modèle d'allocation . . . 68
4.6 Condition de stabilité d'un lisseur de trafic . . . 73

5.1 Répartition des signaux utilisés pour la variable *wpId* . . . 75
5.2 Premiers événements d'un comportement de M_1 avec $\phi_{M_1} = 20$ . . . 78
5.3 Un comportement de la fonction KU_1 . . . 79
5.4 Exemple d'une variable sporadique dépendant de deux variables . . . 80
5.5 Signaux en relation dans le processus P_{VL} . . . 82
5.6 Deux sorties possibles d'un lisseur de trafic pour une même arrivée . . . 82
5.7 Charge de travail et interférence pour l'arrivée (4) . . . 83
5.8 Fonction de travail du lisseur de trafic . . . 84
5.9 Signaux en relation dans le processus d'un port . . . 86
5.10 Prise en compte de la latence technologique d'un port . . . 87
5.11 Distance minimale entre deux occurrences successives en sortie d'un capteur . . . 89
5.12 Distance minimale entre k occurrences successives en sortie d'un capteur . . . 89
5.13 Exemple du processus d'un RDC . . . 90

6.1 Exemple de dépendances étendues . . . 95
6.2 Exemple de latences d'une chaîne . . . 95
6.3 Exemple de fraîcheurs d'une chaîne . . . 97
6.4 Exemple de chaînes divergentes . . . 99
6.5 Exemple de chaînes divergentes avec $F_1 = F_1$ . . . 99
6.6 Exemple de chaînes divergentes de l'étude de cas . . . 99
6.7 Illustration de la cohérence sur chaînes fonctionnelles divergentes . . . 100
6.8 Exemple de chaînes convergentes . . . 101
6.9 Exemple de chaînes convergentes de l'étude de cas . . . 101
6.10 Exemple de mesures de cohérence pour deux chaînes convergentes . . . 102

7.1 Abstraction d'un processus dans le *tagged signal model* . . . 110
7.2 Abstraction d'un ensemble de processus par un autre ensemble de processus . . . 111
7.3 Architecture de l'étude de cas restreinte . . . 115
7.4 Abstraction du réseau de l'étude de cas restreinte . . . 115
7.5 Architecture abstraite de l'étude de cas . . . 117
7.6 Un système illustratif . . . 118
7.7 Graphe des contraintes . . . 120

8.1 Fraîcheur pire cas à l'initialisation et en fonctionnement . . . 124
8.2 Croisement de chaînes fonctionnelles au niveau de f . . . 125
8.3 Illustration du meilleur cas de cohérence . . . 129
8.4 Exemple d'un comportement donnant la latence pire cas de $\mathcal{L}_1$ . . . 132
8.5 Exemple d'un comportement donnant la latence meilleur cas $\mathcal{L}_1$ . . . 132
8.6 Evolution des latences meilleur et pire cas en fonction des meilleurs et pires délais de communication . . . 133
8.7 Exemple d'un comportement donnant la fraîcheur pire cas de $\mathcal{F}_1$ . . . 134
8.8 Evolution de la fraîcheur pire cas en fonction du pire délai de communication . . . 134
8.9 Evolution de la cohérence pire cas entre $\mathcal{L}_1$ et $\mathcal{L}_2$ en fonction du pire délai de communication 135

9.1 Définition d'une étude de cas paramétrée . . . 138
9.2 Temps de résolution des programmes linéaires en nombres entiers pour la latence, la fraîcheur et les cohérences entre chaînes fonctionnelles divergentes et convergentes . . . 139
9.3 Illustration du gain de la méthode . . . 141
9.4 Evolution de L_{max} en fonction de δ^{max} . . . 142
9.5 Extension aux chaînes emboîtées . . . 143

C.1 Notation des indices d'occurrences utilisés . . . 163
C.2 Chaînes divergentes de l'étude de cas . . . 165
C.3 Illustration du meilleur cas de cohérence . . . 166

C.4 Chaînes convergentes de l'étude de cas 167

Liste des tableaux

1.1 Paramètres des partitions de l'étude de cas 10
1.2 Paramètres des liens virtuels de l'étude de cas 11

4.1 Caractéristiques des fonctions de l'étude de cas 64
4.2 Taille et nature des variables du FMS 65
4.3 Caractéristiques des liens virtuels de l'étude de cas 74

7.1 Caractéristiques des canaux temporisés de l'étude de cas 117

8.1 Comparatif des résultats des méthodes *globale* et *locale* 135

9.1 Variation du temps de résolution en fonction de la règle de pivotage utilisée 140
9.2 Variation du temps de résolution pour différents algorithmes de *branch and bound* 140

C.1 Pires et meilleurs temps de traversée des éléments du système abstrait 162

Introduction

Contexte

Les systèmes embarqués avioniques

Dans le domaine de l'aéronautique, les systèmes embarqués ont fait leur apparition durant les années 60 lorsque les équipements analogiques ont commencé à être remplacés par leurs équivalents numériques. Durant la décennie suivante, les premiers équipements informatiques, de faible puissance, ont été embarqués à bord des aéronefs permettant ainsi d'accroître les performances et la capacité de traitement. Dès lors, l'engouement suscité par les progrès de l'informatique fut tel que de plus en plus de fonctionnalités (pilotage, guidage, amortissement du roulis,. . .) ont été numérisées. Ainsi, l'informatique embarquée dans les avions a connu de nombreuses évolutions du fait de l'accroissement de la puissance de calcul nécessaire et de la complexité des systèmes. En effet, depuis les premières architectures centralisées des premiers aéronefs, les systèmes embarqués dans les Airbus A320 et A340 sont organisés autour de fonctions interconnectées disposant de leurs propres ressources. L'architecture de ces systèmes est dite *fédérée*.

Du fait de l'accroissement permanent de la complexité des systèmes, de nouveaux besoins sont apparus durant les années 90, ce qui a conduit à la définition d'une architecture *modulaire* appelée *Avionique Modulaire Intégrée*. A la différence des architectures *fédérées*, les architectures *modulaires* reposent sur le partage des ressources de calcul et de communication entre les fonctions. Ce type d'architecture est appliqué aussi bien dans le domaine civil, avec le Boeing B777 et l'Airbus A380, que dans le domaine militaire, avec l'Eurofighter, le Rafale, le Mirage 2000-9 ou encore l'A400M.

Caractéristiques des systèmes avioniques

Architecture distribuée. L'intégration de l'informatique dans les systèmes embarqués a conduit au développement de deux grands types d'architecture : distribuée à contrôle centralisé et distribuée sans contrôle central. Par exemple, les systèmes spatiaux (satellites, lanceurs,. . .) ou aéronautiques militaires ont longtemps appartenu à la première famille ; le système global étant contrôlé par un (ou deux en cas de redondance), calculateur(s) maître(s) gérant l'exécution et les communications des calculateurs secondaires. A l'inverse, les systèmes avioniques civils ont opté pour des architectures sans contrôleur central. Dans ce type d'architecture, les différentes fonctionnalités du système, telles que la régulation d'un moteur ou la navigation, sont indépendantes les unes des autres. En outre, ces fonctionnalités peuvent être réalisées à l'aide de plusieurs fonctions réparties sur plusieurs calculateurs. Par exemple, la fonctionnalité permettant d'estimer l'heure d'arrivée à un point de navigation, requiert l'utilisation d'une séquence de fonctions : l'acquisition de la pression atmosphérique à l'extérieur de l'aéronef, la conversion de cette donnée en vitesse, la déduction du temps restant pour rejoindre le point de navigation et finalement la gestion de l'affichage. Une telle séquence de fonctions est dénommée *chaîne fonctionnelle*. La notion de *chaîne fonctionnelle* est centrale pour les architectures sans contrôleur central, car la compréhension du fonctionnement du système global passe alors par la compréhension du comportement de chacune des chaînes fonctionnelles et de la façon dont elles interagissent. Cette thèse s'inscrit dans le cadre des systèmes embarqués distribués sans contrôleur central.

Architecture asynchrone. Un second aspect important de ces systèmes réside dans la nature des communications entre les constituants du système, c'est-à-dire entre les différentes fonctions et les composants les exécutant. Pour des raisons de criticité, ces communications sont le plus souvent asynchrones. En d'autres termes, les communications sont réalisées par diffusion de flots de données sans synchronisation

entre un émetteur et le (ou les) récepteur(s). Ce type d'architecture de communication présente l'avantage d'être naturellement robuste aux défaillances : le blocage d'un récepteur, par exemple, n'entraine pas systématiquement le blocage de l'émetteur. A l'inverse, l'inconvénient de cet asynchronisme est que, par définition, les constituants du système ne sont pas synchronisés. Leur comportement global est par conséquent sensible aux diverses latences, gigues et dérives induites par l'exécutif, le matériel ou encore les accès concurrents aux ressources. La prédictibilité de ces phénomènes devient alors impérative. Ce besoin de prédictibilité temporelle se traduit par un ensemble d'*exigences temps réel* que doit satisfaire le système.

Architecture statique. Un troisième trait fréquemment rencontré dans les systèmes embarqués, que nous considérons, réside dans le caractère statique de leur implémentation : l'allocation des fonctions sur les ressources de calcul et l'allocation des ressources de communication sont déterminées hors-ligne et sont constantes durant le fonctionnement du système. L'intérêt majeur de cette approche est d'améliorer la prédictibilité du système et ainsi d'en faciliter l'analyse.

Le problème exploré : la véri cation d'exigences temps réel

Dans cette thèse, nous nous focalisons sur les systèmes ayant une architecture *Avionique Modulaire Intégrée*, donc sans contrôle central, asynchrone et statique. Notre objectif est la vérification d'exigences temps réel sur ces systèmes. Les exigences temps réel s'articulent autour de la notion de *chaîne fonctionnelle*. Une exigence spécifie alors une borne maximale ou minimale pour une propriété temporelle d'une ou plusieurs chaînes fonctionnelles. Par exemple, quel que soit le scénario d'exécution, le temps écoulé entre la sélection d'un point de navigation par le pilote, et sa prise en compte par le système de gestion de vol, doit être inférieur à 700 ms. Autrement dit, dans le pire des scénarios (on utilisera le terme *pire cas* dans la suite) ce temps ne doit pas excéder 700 ms. Nous avons identifié trois types d'exigences temps réel, que nous considérons pertinents vis-à-vis des systèmes étudiés. Il s'agit des exigences de latence, de fraîcheur et de cohérence.

Exigence de latence d une chaîne fonctionnelle. Cette exigence spécifie la durée maximale, ou minimale, acceptable pour la propagation d'un événement le long d'une chaîne fonctionnelle.

Exigence de fraîcheur d une chaîne fonctionnelle. Cette exigence porte sur la notion de fraîcheur des sorties d'une chaîne fonctionnelle, par rapport aux entrées qui ont contribué à son élaboration. Considérons par exemple l'affichage de l'information de vitesse sur un écran de navigation. A un instant t, cette information caractérise la situation réelle de l'avion à l'instant $t - f$, c'est-à-dire l'instant où la vitesse physique de l'avion a été mesurée. Cette durée f est appelée la *fraîcheur* de l'affichage. Elle dépend bien évidemment de la latence de la chaîne, mais également de la période de rafraîchissement de l'ensemble des données en jeu dans la chaîne. Pour faire simple, si $f(t)$ est la fraîcheur de la vitesse à l'instant t, et si à l'instant $t + \delta$ la vitesse n'a pas été mise à jour, alors la fraîcheur à cette instant est : $f(t + \delta) = f(t) + \delta$. Une exigence de fraîcheur spécifie une valeur maximale, ou minimale, acceptable pour la fraîcheur.

Exigence de cohérence entre chaînes fonctionnelles. Enfin, pour des raisons de sûreté de fonctionnement, une fonctionnalité avionique est souvent réalisée par plusieurs chaînes redondantes exécutées en parallèle. La détection de fautes est réalisée par des opérations de comparaison entre les résultats de chaque chaîne. Il est alors important de garantir que ces opérations de comparaison portent sur des données élaborées aux mêmes instants et caractérisant des situations de l'avion temporellement proches. On dit qu'elles doivent être temporellement cohérentes. Une exigence de cohérence spécifie une valeur maximale, ou minimale, pour la cohérence. Par exemple, l'affichage dans le cockpit de l'heure estimée d'arrivée à une destination est élaboré à partir d'une évaluation de la vitesse de l'appareil. La vitesse est évaluée à partir de mesures de la pression atmosphérique. Pour des raisons de sûreté, plusieurs capteurs sont utilisés pour mesurer cette pression et, pour chaque capteur, une information de vitesse est déduite. L'estimation de l'heure d'arrivée est alors effectuée en comparant ces différentes vitesses. Pour que cette comparaison ait du sens, il faut donc garantir que toutes les mesures de pressions utilisées dans une estimation de l'heure d'arrivée aient été prises dans une même fenêtre temporelle.

Contribution et plan du mémoire

Ces trois types d'exigences doivent alors être vérifiés sur un modèle global du système. Nous proposons une modélisation complète d'un système *Avionique Modulaire Intégrée* dans le formalisme du *tagged signal model*. Les exigences temps réel sont également exprimées dans ce formalisme. Nous montrons alors comment, à partir de ce modèle, nous pouvons générer pour chaque exigence, un programme linéaire mixte, c'est-à-dire contenant à la fois des variables entières et des variables réelles, dont la solution optimale permet de vérifier si l'exigence est satisfaite ou pas. Un outil de calcul a été développé pour valider cette méthode. Les résultats obtenus sont probants et montrent que la méthode permet de traiter des systèmes de la taille du système de l'A380. La présentation de ces travaux s'articule en trois parties :

Partie I dédiée au contexte dans lequel s'inscrit cette thèse. Dans un premier temps (chapitre 1), nous présentons le contexte industriel, c'est-à-dire les systèmes *Avioniques Modulaires Intégrées*. Après en avoir rappelé les principes fondamentaux, nous décrivons une étude de cas qui nous permet d'illustrer ces principes. Nous présentons également au travers de cette étude de cas l'objectif de nos travaux, à savoir la vérification d'exigences temps réel telles que des exigences de latence, de fraîcheur et de cohérence. Une fois cet objectif défini, nous dressons un état de l'art correspondant (chapitre 2), qui aborde plusieurs thématiques. Tout d'abord, nous avons cherché à formaliser le problème posé. Nous avons étudié les méthodes de description d'architecture des systèmes embarqués ainsi que la modélisation de leur comportement. Ensuite, nous nous sommes concentrés sur le réseau de communication d'un système avionique modulaire intégrée, qui est un élément particulièrement complexe du système. Nous nous sommes alors intéressés aux méthodes d'évaluation de performance des réseaux embarqués. Les résultats de ces techniques nous serviront par la suite pour définir une abstraction du réseau qui facilitera la vérification des exigences temps réel sur le système global. Nous finissons en passant en revue différentes méthodes permettant l'évaluation de propriétés temps réel des systèmes embarqués. Nous nous sommes efforcés à illustrer cet état de l'art à partir d'exemples tirés de l'étude de cas. En conclusion de cette partie (chapitre 3), nous résumons notre contribution dans ce contexte, ainsi que la démarche que nous avons suivie. Ce chapitre ouvre la voie à la description détaillée des travaux de thèse.

Partie II donne une description formelle du problème que nous explorons. Nous définissons d'abord le modèle *statique* du système (chapitre 4). Notre choix est de représenter l'architecture du système en différentes vues : l'architecture fonctionnelle, l'architecture matérielle et finalement la relation d'allocation qui les lie. Ensuite, nous établissons un modèle *dynamique* du système représentant l'ensemble des comportements possibles du système (chapitre 5). Pour cela, nous modélisons dans le formalisme *tagged signal model* les composants élémentaires du système, dont le comportement est connu, et nous précisons la méthode de composition permettant d'exprimer le comportement du système global. Finalement (chapitre 6), nous exprimons dans ce formalisme les exigences temps réel que le système doit satisfaire. Nous définissons ainsi les fondations de notre méthode de vérification.

Partie III exploite la sémantique du système et des exigences ainsi définies pour vérifier la satisfaction des exigences temps réel. La vérification d'une exigence est ramenée à la résolution d'un programme linéaire mixte (chapitre 7). Le modèle concret du système ne peut être directement utilisé, car il est trop complexe pour être analysé de cette façon. Une approche par abstraction est alors introduite pour limiter la complexité du programme linéaire mixte permettant la vérification d'une exigence. Ensuite (chapitre 8), nous nous intéressons à la mise en œuvre de la résolution de ce programme. Cette méthode est testée sur les exigences de l'étude de cas. Les résultats obtenus sont comparés à ceux obtenus à l'aide de formules analytiques *locales*, c'est-à-dire utilisant les pires et meilleurs cas *locaux* de chaque composant pour déterminer un pire ou meilleur cas global.

Acronymes

A	AADL	: Architecture and Analysis Design Language
	ADIRU	: Air Data Inertial Reference Unit
	AFDX	: Avionics Full-DupleX switched ethernet
	ANR	: Agence Nationale de la Recherche
	API	: Application Programming Interface
	ARINC	: Aeronautical Radio, INCcorporated
B	BAG	: Bandwidth Allocation Gap
	BCCC	: Best Case Consistency of Convergent functional chains
	BCCD	: Best Case Consistency of Divergent functional chains
	BCL	: Best Case Latency
C	CADP	: Construction and Analysis of Distributed Processes
	CAN	: Controller Area Network
	CCSL	: Clock Constraint Specification Language
	CTL	: Computational Tree Logic
E	EBNF	: Extended Backus-Naur Form
	ECU	: Electronic Control Unit
	EDF	: Earliest Deadline First
	ETA	: Estimated Time of Arrival
F	FIFO	: First In First Out
	FM	: Flight Manager
	FMS	: Flight Management System
I	ILP	: Integer Linear Program
	IMA	: Integrated Modular Avionics
K	KCCU	: Keyboard and Cursor Control Unit
	KU	: Keyboard and cursor control Unit

L	LFG	: Logical Functional Group
M	MAF	: MAjor time Frame
	MARTE	: Modeling and Analysis of Real-time and Embedded systems
	MFD	: MultiFunction Display
	MILP	: Mixed Integer Linear Program
	MoC	: Model of Computation
	MTBF	: Mean Time Between Failure
N	NDB	: Navigation DataBase
P	PGCD	: Plus Grand Commun Diviseur
	PPCM	: Plus Petit Commun Multiple
R	RDC	: Remote Data Concentrator
	RTC	: Real-Time Calculus
S	SATRIMMAP	: SAfety and Time cRItical Middleware for future Modular Avionics Platform
	SysML	: System Modeling Language
T	TDMA	: Time Division Multiple Access
	TCTL	: Timed Computational Tree Logic
	TLA	: Temporal Logic of Action
	TSM	: Tagged Signal Model
V	VHF	: Very High Frequency
	VL	: Virtual Link
	VOR	: VHF Omnidirectional Range
W	WCCC	: Worst Case Consistency of Convergent functional chains
	WCCD	: Worst Case Consistency of Divergent functional chains
	WCF	: Worst Case Freshness
	WCL	: Worst Case Latency
	WCET	: Worst Case Execution Time
	WCTT	: Worst Case Traversal Time

Notations

Architecture du système

S	: le système
$Arch_F$	: l'architecture fonctionnelle du système
$Arch_M$	: l'architecture matérielle du système
$Alloc$	: l'allocation de l'architecture fonctionnelle sur l'architecture matérielle
$Func$	: l'ensemble des fonctions avioniques
Var	: l'ensemble des variables échangées entre les fonctions
Dep	: l'ensemble des dépendances entre les variables
$v.size$	: la taille de la variable v
$v.nat$	: la nature de la variable v
$f.Input$	: l'ensemble des variables en entrée de la fonction f
$f.Output$	: l'ensemble des variables en sortie de la fonction f
T_f	: la période de la fonction f
C_f	: la durée maximale d'exécution de la fonction f
$Ports$	: l'ensemble des ports de communication
$Modules$	: l'ensemble des modules
RDC	: l'ensemble des RDC
$Switches$	: l'ensemble des commutateurs AFDX
$Links$	: l'ensemble des liens Ethernet
$Sensors$	: l'ensemble des capteurs
$Actuators$	: l'ensemble des actionneurs
$p_1.d$	: le débit du port p_1
$M.port$	: le port du module M
$sw.ports$	: les ports du commutateur sw
$sw.lag$	: la latence technologique du commutateur sw
$s.nat$	: la nature du capteur s
T_s	: la période d'échantillonage ou le temps minimum entre deux échantillons du capteur s
a_s	: le temps minimal de traversée du bus du capteur s
b_s	: le temps maximal de traversée du bus du capteur s
a_A	: le temps minimal de traversée du bus de l'actionneur A
b_A	: le temps maximal de traversée du bus de l'actionneur A
$EtePath$	: l'ensemble des chemins du réseau
$p.start$	: le premier port du chemin p
$p.end$	: le dernier port du chemin p
Var_S	: l'ensemble des variables sources
Var_P	: l'ensemble des variables puits
Var_I	: l'ensemble des variables intermédaires
$Alloc_F$	: la fonction d'allocation des fonctions
$Alloc_S$	: la fonction d'allocation des variables sources
$Alloc_P$	: la fonction d'allocation des variables puits
$Alloc_D$	: la fonction d'allocation des dépendances
$vl.s_{min}$	: le taille minimale d'une trame du lien virtuel vl
$vl.s_{max}$	: le taille maximale d'une trame du lien virtuel vl
$vl.paths$	: l'ensemble des chemins couverts par le lien virtuel vl
$varVL(vl)$	: l'ensemble des variables allouées au lien virtuel vl
N_{vl}	: le nombre maximale de trames transmises par le lien virtuel à chaque exécution

Modèle comportemental *tagged signal model*

$\mathbb{S}$	: l'ensemble des comportements possibles du système S
P_e	: le processus de l'élément e
$\mathbf{s}$	: un vecteur de signaux, i.e. un comportement
s_f	: le signal d'activation de la fonction f
s_{x_e}	: le signal de la variable x en sortie de l'élément e
$s_j[i]$	: le $i^{\text{ème}}$ événement du signal s_j
t_i^j	: l'estampille de l'événement $s_j[i]$
v_i^j	: la valeur de l'événement $s_j[i]$
$v_{i\ k}^j$	: la composante du signal s_k de la valeur de l'événement $s_j[i]$
	: valeur signifiant une absence de dépendance
$HP(m)$	: l'hyper-période du module m
ϕ_e	: la phase de l'équipement e
$W(t)$	: charge de travail à l'instant t
$I(v,n)$	: interférence subie par la $n^{\text{ième}}$ occurrence de la variable v
$n_I(t)$	: nombre d'occurrences de variables arrivant exactement à l'instant t

Divers

$\mathbb{N}$	: l'ensemble des entiers naturels
$a..b$	: l'ensemble des entiers compris entre a et b inclus
$\mathbb{R}^+$	: l'ensemble des nombres réels positifs
t.q.	: tel(s,le,les) que

Première partie

Contexte industriel et scienti que

Introduction de la première partie

Cette première partie est dédiée au contexte dans lequel s'inscrit cette thèse. Dans un premier temps (chapitre 1), nous présentons le contexte industriel, c'est-à-dire les systèmes avioniques modulaires intégrées. Après en avoir rappelé les principes fondamentaux, nous décrivons une étude de cas qui nous permet d'illustrer ces principes. Nous présentons également au travers de cette étude de cas l'objectif de nos travaux, à savoir la vérification d'exigences temps réel telles que des exigences de latence, de fraîcheur et de cohérence.

Une fois cet objectif défini, nous dressons un état de l'art correspondant, qui aborde plusieurs thématiques (chapitre 2). Tout d'abord, nous avons cherché à formaliser le problème posé. Nous avons étudié les méthodes de description d'architecture des systèmes embarqués ainsi que la modélisation de leur comportement. Ensuite, nous nous sommes concentrés sur le réseau de communication d'un système avionique modulaire intégrée, qui est un élément particulièrement complexe du système. Nous nous sommes alors intéressés aux méthodes d'évaluation de performance des réseaux embarqués. Les résultats de ces techniques nous serviront par la suite pour définir une abstraction du réseau qui facilitera la vérification des exigences temps réel sur le système global. Nous finissons en passant en revue différentes méthodes permettant l'évaluation de propriétés temps réel des systèmes embarqués. Nous nous sommes efforcés à illustrer cet état de l'art à partir d'exemples tirés de l'étude de cas.

En conclusion de cette partie (chapitre 3), nous résumons notre contribution dans ce contexte, ainsi que la démarche que nous avons suivie. Ce chapitre ouvre la voie à la description détaillée des travaux de thèse.

Chapitre 1

Avionique Modulaire Intégrée

Dans ce chapitre, nous introduisons les principes fondamentaux des systèmes avioniques modulaires intégrés (appelées IMA pour *Integrated Modular Avionics*). Ces systèmes constituent le cadre dans lequel les résultats de cette thèse sont appliqués. Nous illustrons les principes des systèmes IMA avec une étude de cas tirée d'un système de gestion de vol, et fournie dans le cadre du projet ANR SATRIMMAP. Cette étude de cas sera également le support avec lequel nous illustrerons les notions abordées dans le manuscrit.

1.1 Architecture avionique modulaire intégrée

Les architectures avioniques modulaires intégrées sont apparues dans les années 90 pour exécuter un ensemble d'applications partageant des ressources de calcul, appelées modules, et communiquant par un réseau partagé lui-même connecté à un ensemble de capteurs/actionneurs embarqués par l'intermédiaire de passerelles appelées RDC (pour *Remote Data Concentrator*). Les modèles d'exécution et de communication des architectures IMA sont définis par les standards ARINC653 [12] et ARINC664 (partie 7) [13] adoptés par Airbus depuis l'A380 et par Boeing depuis le B777. L'objectif de ces standards est d'assurer un niveau élevé de sûreté de fonctionnement, en particulier en garantissant à la fois la ségrégation des fonctions avioniques et un certain degré de prédictibilité temporelle, c'est-à-dire une capacité de preuve de la satisfaction d'un ensemble d'exigences temps réel. L'ARINC653 définit les règles de partage des modules entre les fonctions. L'ARINC664 (partie 7), aussi connu sous le nom d'AFDX (pour *Avionics Full-DupleX switched ethernet*), définit les règles de partage du réseau de communication inter-modules.

Ce chapitre n'a pas vocation à être une présentation exhaustive de ces standards, en revanche nous nous concentrons sur les propriétés pertinentes vis à vis de notre modèle et de nos objectifs. En particulier, nous décrivons plus en détail la notion de module IMA et le réseau AFDX.

1.1.1 Module IMA

Le concept IMA repose sur la séparation du développement des ressources et des fonctions avioniques. Les ressources de calcul sont des *modules* interconnectés par le réseau AFDX et potentiellement connectés à d'autres équipements, comme des capteurs ou des actionneurs, par différents moyens de communication tels que des bus de terrain. Pour garantir la ségrégation des fonctions avioniques, la notion de *partition* est utilisée.

Partition

Chaque fonction s'exécute au sein d'une partition. Une partition est caractérisée par une zone mémoire qui lui est propre et une tranche de temps, répétée périodiquement, pendant laquelle elle est la seule à pouvoir s'exécuter sur le module. Le partage des ressources de calcul du module est donc réalisé grâce à un découpage temporel. Les partitions d'un module sont ordonnancées statiquement et de façon strictement périodique : si une occurrence d'une partition démarre à un instant t et que la partition a une période T, alors le prochain démarrage de la partition aura lieu à l'instant $t + T$. L'ordonnancement de toutes les partitions d'un module est décrit par une trame cyclique, appelée la MAF (pour *MAjor time Frame*). Comme les périodes de ces partitions ne sont pas nécessairement identiques, la longueur de la MAF est égale à l'hyper-période des partitions du module, c'est-à-dire le *ppcm* (pour *plus petit commun multiple*)

des périodes des partitions. Les partitions sont alors caractérisées par leur période, leur durée et un décalage relatif au démarrage de la MAF, appelé *o set*. Ces notions sont représentées sur la figure 1.1. Les fonctions F_1 et F_2 de période T_1 et T_2 s'exécutent dans les partitions représentées par les boîtes notées F_1 et F_2. O_2 correspond à l'*o set* de la partition de F_2. L'*o set* de F_1 coïncide avec le début de la MAF, il est donc nul et n'est pas représenté.

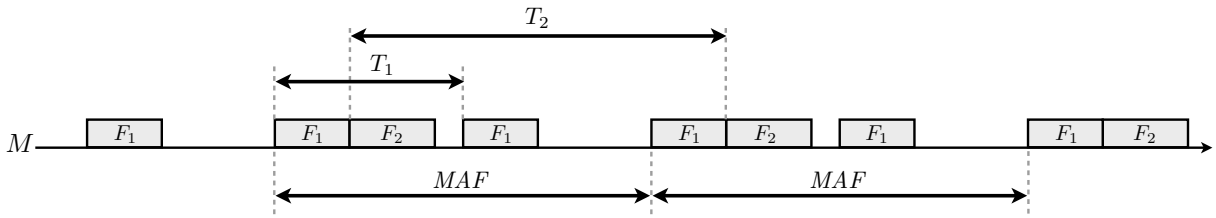


FIGURE 1.1 – Exemple d'une MAF

A noter que l'ARINC 653 permet d'attribuer plusieurs tranches de temps à une partition. Dans un souci de simplicité, nous nous restreignons à des partitions avec une unique tranche de temps.

A l'intérieur d'une partition, la fonction avionique est réalisée par un ensemble de processus qui sont ordonnancés localement à la partition : les processus de F_1 peuvent par exemple être ordonnancés avec une politique à priorité fixe tandis que les processus de F_2 peuvent être ordonnancés avec une politique EDF (pour *Earliest Deadline First*). Dans cette thèse, nous ne prenons pas en compte ce niveau de détail. Une fonction sera seulement caractérisée par son temps d'exécution pire cas (WCET pour *Worst Case Execution Time*).

Communication

Les communications entre les partitions s'effectuent au travers de ports de communication, dit ports APEX, qui peuvent être de type :
- *sampling* où seule la dernière donnée reçue est conservée,
- *queuing* où le port se comporte comme une file d'attente.

Les ports APEX sont unidirectionnels : ils sont utilisés soit en lecture, soit en écriture. Dans la suite, nous ferons l'hypothèse qu'une fonction lit l'ensemble de ses ports d'entrée au démarrage de sa partition et réalise l'ensemble des écritures sur ses ports de sortie à la fin de son traitement et avant la fin de la partition.

Remote Data Concentrator

Un RDC est un module spécialisé qui relie des équipements sans connexion AFDX, tels que des capteurs ou des actionneurs, à la plateforme IMA. La fonctionnalité d'un RDC est donc de convertir des données provenant d'une technologie de type *bus* (bus de terrain, ARINC429, CAN,...) vers l'AFDX et réciproquement. La transformation consiste à traduire les données dans le format adéquat en modifiant leur encodage par exemple. Les RDC peuvent être considérés comme des passerelles sans OS dont tous les ports sont de type *sampling*.

1.1.2 Réseau AFDX

Le réseau AFDX est dérivé de l'Ethernet commuté. Ce réseau est composé principalement de commutateurs, qui sont les éléments clefs de l'architecture et de producteurs/consommateurs de données, qui sont appelés les abonnés. Dans cette thèse, les abonnés que nous considérons sont les modules IMA et les RDC. Le réseau doit :
- garantir la fiabilité des données.
- offrir une disponibilité élevée. De nombreux équipements étant reliés au réseau, il est prévu un temps moyen entre panne (MTBF pour *Mean Time Between Failure*) très élevé, de l'ordre de 100000 heures.
- garantir l'ordre des paquets au sein d'un même flux, ainsi qu'un chemin unique de bout en bout pour tous ces paquets, c'est-à-dire que le routage des paquets doit être statique.

- fournir une isolation des erreurs, entre éléments consécutifs ou entre flux.
- fournir un service déterministe de transfert de données. La latence maximale de bout en bout, ainsi que la gigue, doivent être connues. Aucune perte de trame par congestion n'est tolérée.

Ce dernier point est très important dans le processus de validation du système avionique. En effet, dans le cas d'un réseau Ethernet commuté *Full-Duplex*, le risque de perte de trames est présent. L'utilisation de liens *Full-Duplex* supprime le risque de collision sur les liens physiques. En revanche, les files d'attente internes aux commutateurs sont de taille limitée, un trafic trop important pourrait les faire déborder. En outre, l'occupation des files d'attente entraîne des délais dans les communications. La spécificité du réseau AFDX est la mise en place de mécanismes de contrôle de flux permettant de maîtriser l'utilisation des files d'attentes, et donc la qualité de service, offerte par le réseau. Dans l'AFDX, cette maîtrise de la qualité de service est assurée par l'implémentation de la notion de liens virtuels (VL pour *Virtual Link*).

Lien Virtuel

Cette notion est centrale dans le réseau AFDX. La principale motivation derrière cette notion est de fournir une ségrégation des flux : le mauvais comportement d'un flux ne doit pas nuire aux autres flux. En d'autres termes, un lien virtuel permet de « virtualiser » un bus avionique classique pour chaque flux, où il serait le seul flux à émettre. Cette « virtualisation » a pour objectif de garantir à chaque flux une certaine bande passante, ainsi que des caractéristiques temps réel comme les temps de traversée du réseau pire et meilleur cas, quel que soit le comportement des autres flux. Un lien virtuel est défini par :

- un identifiant unique,
- une ou plusieurs adresses de destination,
- le chemin emprunté pour rallier ces destinations,
- la taille minimale et maximale d'un paquet,
- un temps minimum d'émission entre deux paquets consécutifs, appelé *bag* (pour *Bandwidth Allocation Gap*).

Un émetteur ne peut donc émettre sur un VL qu'au maximum un paquet de taille maximale tous les *bag*. En notant s_{max} cette taille maximale, le débit maximal d'un VL, noté ρ, est donc :

$$\rho = \frac{s_{max}}{bag} \tag{1.1}$$

Dans la suite, nous appelons *lisseur de trafic* le mécanisme en entrée d'un VL, forçant le temps minimum d'émission entre deux paquets. Nous représentons son fonctionnement sur la figure 1.2. A fin de chaque exécution, la fonction F_1 produit des données qu'elle place dans les ports APEX correspondants. Sur cet exemple, 4 messages sont produits et placés dans 3 ports APEX, et ces ports émettent dans un même VL. Ensuite, ces messages sont placés dans la file d'attente du VL. Le lisseur de trafic cadence alors ces messages au rythme d'un tous les *bag*. La politique de service est « premier arrivé premier servi » (FIFO pour *First In First Out*). Le flux en sortie lisseur de trafic est alors mélangé aux autres VL produits par F_1 ou par les autres fonctions du module. L'ordonnanceur des VL sera supposé utiliser une politique de service FIFO. L'agrégation de ces flux est alors transmise sur le réseau via le port Ethernet du module. Du fait de ce mélange, les paquets d'un VL ne sont potentiellement plus cadencés tous les *bag*, comme représentés sur la figure.

Conclusion

Nous avons présenté les principes fondamentaux des systèmes IMA, à savoir la mutualisation des ressources de calcul et de communication. Dans la suite, nous illustrons ces principes sur une étude de cas tirée d'un système de gestion de vol. Une description plus détaillée de la mise en œuvre de ces principes sera réalisée au chapitre 4, où nous présentons un modèle d'architecture pour ces systèmes.

1.2 Etude de cas : un système de gestion de vol

Dans cette section, nous introduisons l'étude de cas qui sera utilisée pour illustrer les différentes notions du mémoire. Il s'agit d'une sous-partie du système de navigation (FMS pour *Flight Management System*) dont l'objectif est de contrôler l'affichage d'informations de navigation sur les écrans de pilotage. Nous présentons tout d'abord les fonctions avioniques impliquées dans la réalisation du FMS, ensuite l'architecture sur laquelle ces fonctions sont déployées et finalement les exigences de latence, fraîcheur

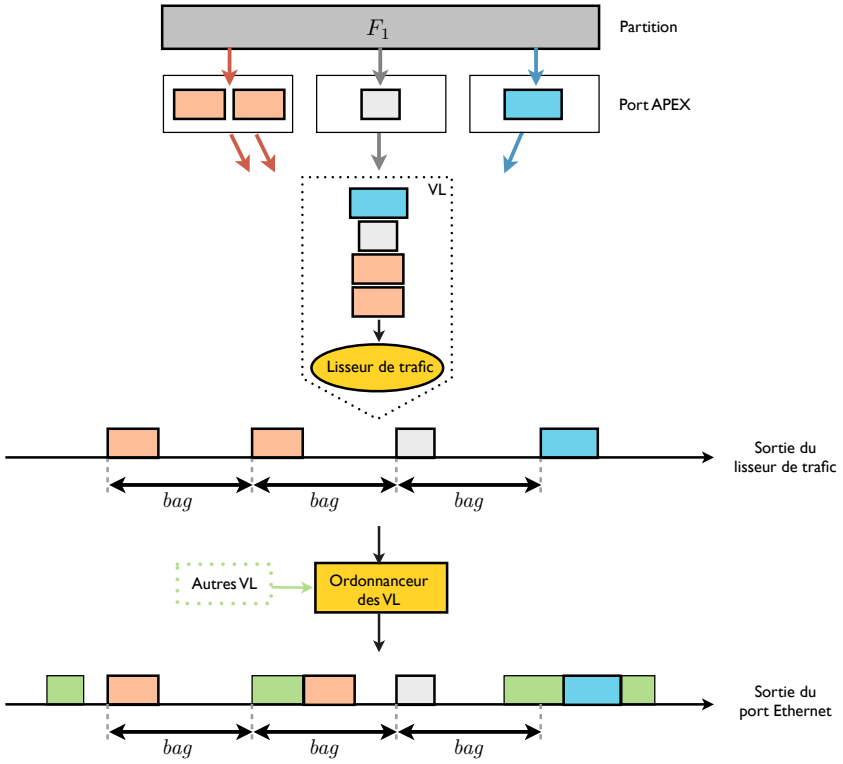


FIGURE 1.2 – Fonctionnement d'un lisseur de trafic

et cohérence que doit satisfaire le système. La description formelle des fonctions et de l'architecture est réalisée au chapitre 4 (page 61). La formalisation des exigences est présentée au chapitre 6 (page 93).

Les différentes notions d'exigence ainsi que le fonctionnement du système sont illustrés à l'aide d'exemples d'exécution.

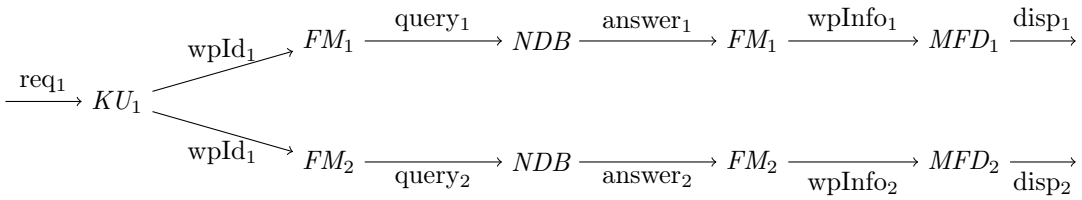
1.2.1 Fonctions

Le système de navigation interagit avec l'équipage au travers d'écrans et de claviers. Sur requête du pilote ou du copilote, entrée au moyen de leur clavier respectif, le système doit afficher les informations du prochain point de navigation (on utilisera le terme commun *waypoint* dans la suite du mémoire) sur les écrans. Ces informations sont de deux types : des informations statiques sur le *waypoint* telles que les coordonnées du *waypoint* et la fréquence VOR qui permet de déterminer le cap en direction du *waypoint*, et des informations dynamiques mises à jour périodiquement telles que l'heure estimée d'arrivée (notée ETA pour *Estimated Time of Arrival*).

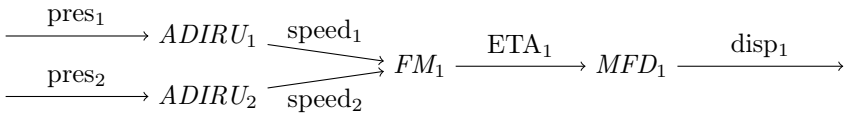
Pour des raisons de sûreté, le FMS repose sur une architecture redondante (composée de deux voies notées 1 et 2). Sur chaque voie i, le FMS est composé d'une fonction KU_i (pour *Keyboard and cursor control Unit*) contrôlant le clavier et d'une fonction MFD_i (pour *Multi Functional Display*) gérant l'affichage des pages contenant les informations de navigation. Le pilote peut entrer une requête d'affichage d'un *waypoint*. Cette requête est reçue par la fonction KU_i ($i = 1$ pour le pilote, $i = 2$ pour le copilote). La requête est alors transmise aux gestionnaires de vol FM_1 et FM_2 (FM pour *Flight Manager*) qui questionnent en parallèle la base de données de navigation (NDB pour *Navigation DataBase*). Celle-ci retourne aux FM les informations statiques du *waypoint* qui sont ensuite périodiquement enrichies par chaque *FM* avec des informations dynamiques calculées en fonction des données de vol (vitesse, position,...) produites par les centrales inertielles (ADIRU pour *Air Data Intertial Reference Unit*). Au final, chaque FM_i envoie périodiquement à chaque MFD_i ces informations à afficher. Chaque ADIRU élabore les données de vol à partir de données de base fournies par des mesures prises par des capteurs.

Nous décrivons cette architecture fonctionnelle sous la forme de chaînes fonctionnelles telles que représentées sur la figure 1.2.1. La figure 1.3(a) correspond aux chaînes impliquées dans l'affichage des informations d'un *waypoint* suite à une requête du pilote, tandis que la figure 1.3(b) correspond aux

chaînes impliquées dans la production de l'heure estimée d'arrivée (ETA pour *Estimated Time of Arrival*) par le FM_1. Le système étant symétrique sur les deux voies, nous ne représentons ni les chaînes traitant les requêtes du copilote, ni les chaînes générant l'affichage de l'heure estimée d'arrivée pour le copilote.



(a) Réponse à une requête du pilote



(b) Production de l'heure estimée d'arrivée à destination du pilote

FIGURE 1.3 – Chaînes fonctionnelles de l'étude du cas

Les différentes variables échangées entre ces fonctions sont :
- req_i : la requête du pilote ou du copilote,
- $wpId_i$: l'identifiant du *waypoint* choisi par le pilote ou le copilote,
- $query_i$: la requête transmise à la *NDB* par FM_i pour récupérer les informations du *waypoint*,
- $answer_i$: la réponse transmise à FM_i par la *NDB* contenant les informations du *waypoint*,
- $wpInfo_i$: les informations du *waypoint* transmises entre FM_i et MFD_i,
- $disp_i$: la page affichée par MFD_i regroupant les informations statiques du *waypoint* ainsi que l'heure estimée d'arrivée ETA_i,
- $pres_i$: la valeur de la pression atmosphérique à l'extérieur de l'appareil,
- $speed_i$: la valeur de la vitesse de l'appareil déduite de la pression extérieure,
- ETA_i : l'heure estimée d'arrivée au *waypoint*,

La chaîne fonctionnelle $^{pres_1}\ ADIRU_1\ ^{speed_1}\ FM_1\ ^{ETA_1}\ MFD_1\ ^{disp_1}$ se lit de la façon suivante : la fonction $ADIRU_1$ utilise la variable $pres_1$ pour déterminer $speed_1$ qui est transmise à la fonction FM_1. Cette variable est utilisée dans le calcul d'ETA_1 qui est prise en compte par la fonction MFD_1 pour générer la variable $disp_1$. Ces fonctions sont alors déployées sur une architecture IMA que nous décrivons dans la suite.

1.2.2 Architecture

Dans cette section, nous présentons une vue générale de l'architecture IMA sur laquelle est déployé le système de gestion de vol. L'ensemble des paramètres de l'architecture sera détaillé dans le chapitre 4 où nous formaliserons la description d'une telle architecture. A noter que le système présenté est un sous-système de l'appareil et donc que les ressources de calcul et de communication de cette architecture sont partagées avec d'autres fonctions. Cependant, nous ne décrivons que les éléments relatifs au système de gestion de vol FMS.

L'architecture du système de gestion de vol est composée d'un ensemble de modules de calcul interconnectés au travers d'un réseau AFDX. Cette architecture est représentée sur la figure 1.4. Sept modules, de Module 1 à Module 7 (nommés $M_1, \ldots, M_7$ dans la suite), sont utilisés pour héberger les fonctions avioniques. Le réseau AFDX est constitué de cinq commutateurs $S_1, \ldots, S_5$ représentés par les boîtes violettes sur la figure. Les ports des commutateurs ainsi que les ports des modules fonctionnent à 100 Mbits/s.

Les capteurs de pression $sensor_1$ et $sensor_2$ sont reliés aux centrales inertielles $ADIRU_1$ et $ADIRU_2$ au travers des *Remote Data Concentrators* RDC_1 et RDC_2. Les écrans $display_1$ et $display_2$ ainsi que les claviers key_1 et key_2 sont situés dans le cockpit et sont directement reliés aux modules M_1 et M_2 via un bus de terrain.

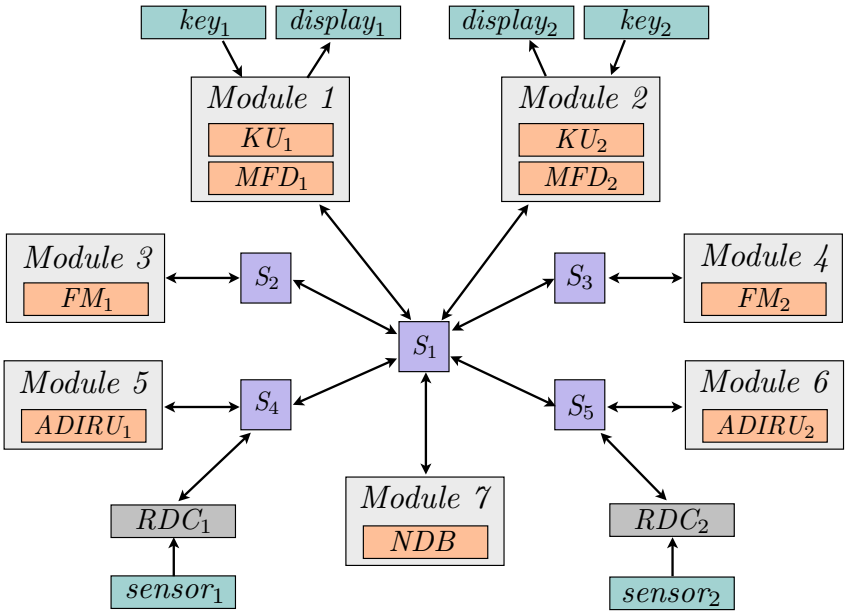


FIGURE 1.4 – Architecture de l'étude de cas

Les modules hébergent des fonctions qui s'exécutent périodiquement. Conformément au standard ARINC 653, ces exécutions ont lieu au sein de partitions afin de garantir la ségrégation des fonctions. Ainsi, chaque partition définit une tranche de temps qui lui est propre, c'est-à-dire qu'aucune autre partition ne peut s'exécuter durant cet intervalle de temps. Cet ordonnancement est statique et est défini hors-ligne. Les tranches de temps sont caractérisées par une période, une durée et un décalage relatif au démarrage de la MAF appelé *o set*. Ces notions ont été présentées à la sous-section 1.1.1 (page 5). Le tableau 1.1 regroupe les paramètres des partitions utilisées dans l'étude de cas. Ces paramètres sont illustrés sur la figure 1.5 qui montre l'ordonnancement statique des partitions du module M_1.

partition	période	durée	offset	module
KU_1	50	25	0	1
MFD_1	50	25	25	1
KU_2	50	25	0	2
MFD_2	50	25	25	2
FM_1	60	30	0	3
FM_2	60	30	0	4
$ADIRU_1$	60	30	0	5
$ADIRU_2$	60	30	0	6
NDB	100	20	0	7

TABLE 1.1 – Paramètres des partitions de l'étude de cas

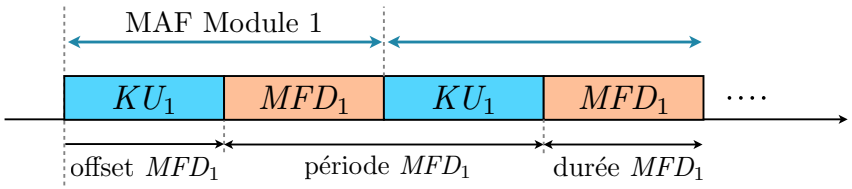


FIGURE 1.5 – Ordonnancement des partitions du module 1

Comme présentés à la sous-section 1.1.2 page 7, les échanges de variables entre fonctions s'effectuent au travers de liens virtuels (VL pour *Virtual Links*). Un VL définit une connexion multipoint entre une source et une ou plusieurs destinations, les sources et les destinations étant des partitions. Par exemple, la fonction KU_1 utilise le lien VL_1 pour communiquer avec FM_1 et FM_2. En entrée de chaque VL un mécanisme de lissage de trafic est implémenté. Il garantit une distance minimale entre chaque émission de message sur le VL, ce qui permet d'éviter l'engorgement du réseau. Cette distance minimale est appelée *BAG* (pour *Bandwidth Allocation Gap*) du VL. Le tableau 1.2 regroupe les paramètres des liens virtuels utilisés dans l'étude de cas. Les variables empruntant chaque VL sont précisées et le chemin du VL correspond à la séquence des commutateurs traversés par le VL. Si un VL a plusieurs destinations alors il peut exister plusieurs chemins pour ce VL.

lien virtuel	source	destination(s)	variable(s)	BAG (ms)	chemin(s)
VL_1	KU_1	FM_1, FM_2	$wpId_1$	32	S_1, S_2 , S_1, S_3
VL_2	KU_2	FM_1, FM_2	$wpId_2$	32	S_1, S_2 , S_1, S_3
VL_3	FM_1	MFD_1	$wpInfo_1$, ETA_1	8	S_2, S_1
VL_4	FM_1	NDB	$query_1$	16	S_2, S_1
VL_5	FM_2	MFD_2	$wpInfo_2$, ETA_2	8	S_3, S_1
VL_6	FM_2	NDB	$query_2$	16	S_3, S_1
VL_7	NDB	FM_1	$answer_1$	64	S_1, S_2
VL_8	NDB	FM_2	$answer_2$	64	S_1, S_3
VL_9	RDC_1	$ADIRU_1$	$pres_1$	32	S_4
VL_{10}	RDC_2	$ADIRU_2$	$pres_2$	32	S_5
VL_{11}	$ADIRU_1$	FM_1, FM_2	$speed_1$	32	S_4, S_1, S_2 , S_4, S_1, S_3
VL_{12}	$ADIRU_2$	FM_2, FM_1	$speed_2$	32	S_5, S_1, S_3 , S_5, S_1, S_2

TABLE 1.2 – Paramètres des liens virtuels de l'étude de cas

Pour des raisons de sûreté du système, ce système doit satisfaire un ensemble d'exigences temps réel que nous présentons dans la suite.

1.2.3 Exigences

Pour illustrer les exigences que doit satisfaire le système, nous en présentons quatre.

Exigence de latence relative à une chaîne fonctionnelle : elle permet de garantir que le système répond suffisamment rapidement à une sollicitation,

E_1 : l'écran du pilote doit afficher les informations d'un *waypoint* au plus tard 700 ms après la saisie du pilote.

Exigence de fraîcheur relative à une chaîne fonctionnelle : elle permet de garantir qu'une variable du système dépend d'une variable suffisamment récente pour être pertinente,

E_2 : l'heure estimée d'arrivée affichée sur l'écran du pilote doit dépendre d'une valeur de pression qui a été mesurée au pire 400 ms plus tôt.

Exigence de cohérence sur chaînes fonctionnelles divergentes : elle permet de garantir qu'un événement en entrée du système génère plusieurs événements arrivant dans une même fenêtre temporelle,

E_3 : les écrans du pilote et du copilote doivent afficher les informations du *waypoint* saisi par le pilote dans une fenêtre de 500 ms. Autrement dit, la différence entre la latence d'une requête du pilote et l'affichage des informations sur son écran et de la latence de cette même requête et l'affichage sur l'écran du copilote doit être inférieure à 500 ms.

Exigence de cohérence sur chaînes fonctionnelles convergentes : elle permet de garantir que la valeur d'une variable dépend de valeurs mises à jour dans une même fenêtre temporelle,

E_4 : l'heure estimée d'arrivée est déterminée à l'aide des valeurs de pression issues de deux capteurs différents. Pour que cette estimation ait du sens, il est nécessaire que ces valeurs soient mesurées dans un intervalle de temps de 300 ms.

Ces exigences ont leur équivalent pour le copilote, mais le système étant symétrique, nous nous contenterons de vérifier les exigences relatives au pilote. A présent, nous précisons ces exigences et le fonctionnement du système à l'aide d'exemples d'exécution.

Exigence de latence

Considérons la chaîne fonctionnelle impliquée dans l'exigence E_1, c'est-à-dire utilisée entre la saisie d'un *waypoint* par le pilote et l'affichage des informations de ce *waypoint* sur son écran : $\overset{req_1}{\rightarrow} KU_1 \overset{wpId_1}{\rightarrow} FM_1 \overset{query_1}{\rightarrow} NDB \overset{answer_1}{\rightarrow} FM_1 \overset{wpInfo_1}{\rightarrow} MFD_1 \overset{disp_1}{\rightarrow}$. L'exigence de latence se comprend de la façon suivante : le temps écoulé entre une occurrence de req_1 et la première occurrence de $disp_1$ qui en dépend doit être au maximum 700 ms. Les occurrences d'une variable correspondent à ses mises à jour. Pour illustrer cette exigence, considérons le scénario représenté sur la figure 1.6.

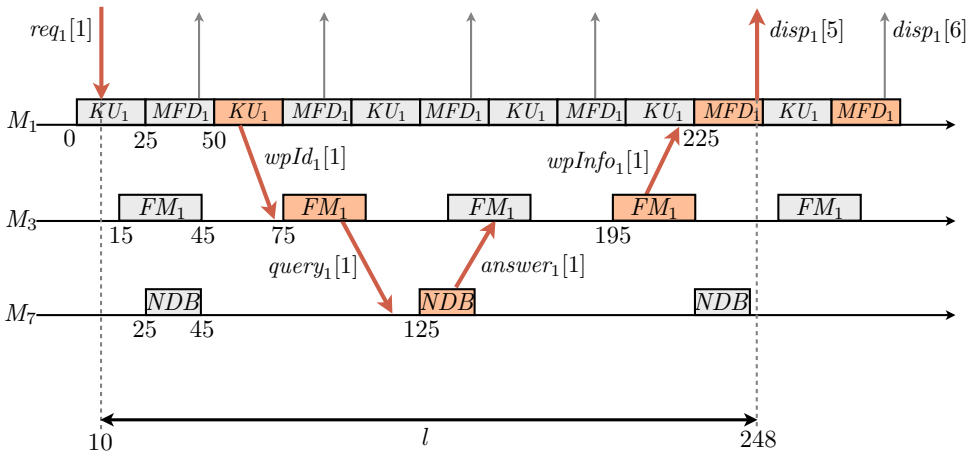


FIGURE 1.6 – Latence d'une requête du pilote

Sur cette figure, nous suivons le traitement de la première requête du pilote, notée $req_1[1]$, jusqu'à l'affichage dans le cockpit des informations du *waypoint* correspondant. Les occurrences des fonctions mises en valeur correspondent à celles traitant la première requête du pilote. Pour faciliter la lecture de l'exemple, l'influence du capteur de saisie key_1 et de l'actionneur réalisant l'affichage $display_1$ ainsi que les détails des communications entre fonctions ne sont pas représentés.

Les modules d'un système IMA sont asynchrones, ce qui signifie qu'ils peuvent démarrer avec un certain déphasage. Dans ce scénario, le module M_1 démarre à l'instant 0, M_3 et M_7 respectivement 15 et 25 ms plus tard. Le pilote émet une requête à 10 ms pour avoir des informations sur un *waypoint*. Comme cette requête arrive après le démarrage de la première occurrence de la fonction KU_1, elle est prise en compte par la deuxième occurrence de la fonction, c'est-à-dire celle qui commence à 50 ms. La requête est traitée par KU_1 et une occurrence de la variable $wpId_1$, notée $wpId_1[1]$, est transmise à FM_1 via VL_1. Avant d'atteindre le module M_3, cette occurrence subit un retard dans le lisseur de trafic du VL, puis dans le réseau le long du chemin du VL. Ensuite, cette occurrence est traitée par la première occurrence de FM_1 démarrant après son arrivée dans le module M_3. Dans ce scénario, il s'agit de la deuxième occurrence de FM_1 commençant à 75 ms. La requête $query_1[1]$ est alors transmise à NDB via le VL_4. La réponse de NDB est transmise à FM_1 (via VL_7), complétée puis transmise MFD_1 (via VL_3). Cette fonction produit un nouvel affichage à chaque exécution, ce qui correspond à la mise à jour de la variable $disp_1$. En particulier, l'affichage est produit en fonction de la dernière occurrence de $wpInfo_1$ reçue. La première occurrence de la variable $disp_1$ contenant les informations du *waypoint* est alors celle produite lors de la cinquième exécution de MFD_1 c'est-à-dire $disp_1[5]$ qui est produite à 248 ms. Dans ce scénario, la latence entre la requête et l'affichage est égale à $l = 248 - 10 = 238$ ms. A noter que l'affichage suivant (qui correspond à l'occurrence $disp_1[6]$) contient les mêmes informations car le pilote n'a pas saisi de nouvelle requête. La latence est définie pour mesurer la réactivité du système et donc cette occurrence n'est pas prise en compte dans la mesure de la latence de la requête, nous ne nous intéressons qu'à la première utilisation de la requête.

Ce scénario n'est cependant pas unique. Plusieurs scénarios sont possibles, conséquence directe de

l'asynchronisme de la plate-forme. Cette variabilité entre les scénarios provient principalement de trois facteurs : (1) l'asynchronisme entre les modules revenant à considérer que les modules démarrent indépendamment les uns des autres avec une phase relative quelconque ; (2) les instants de production des variables par les fonctions peuvent varier à l'intérieur des tranches de temps de la fonction ; et (3) la variabilité des temps de traversée du réseau due à la dynamique du trafic. En conséquence, pour une même exécution, les requêtes successives du pilote ($req_1[2]$, $req_1[3]$, ...) ne subissent pas nécessairement la même latence. L'exigence spécifie que, quelle que soit l'exécution et quelle que soit la requête, la latence d'une requête à un affichage doit toujours être inférieure à 700 ms.

Exigence de fraîcheur

De la même manière, nous illustrons l'exigence de fraîcheur. La pire fraîcheur de l'heure estimée d'arrivée affichée par MFD_1 doit être inférieure à 400 ms. Autrement dit, toute heure estimée d'arrivée affichée sur l'écran par le MFD doit refléter une situation vieille de moins de 400 ms. Pour faciliter la lecture de l'exemple, nous nous limitons à la sous-chaîne $\overset{pres_1}{\rightarrow} ADIRU_1 \overset{speed_1}{\rightarrow} FM_1 \overset{ETA_1}{\rightarrow}$, et considérons le scénario représenté sur la figure 1.7.

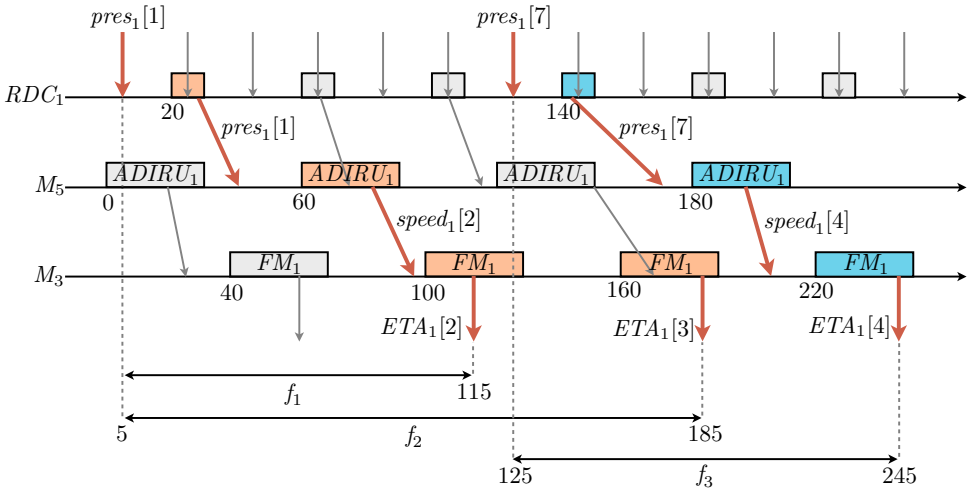


FIGURE 1.7 – Fraîcheur de l'heure estimée d'arrivée

Sur cette figure, nous représentons le lien entre les occurrences de l'heure estimée d'arrivée produites par FM_1 et les mesures de pression délivrées par le capteur $sensor_1$. On observe que les occurrences $ETA_1[2]$ et $ETA_1[3]$ dépendent de la même mesure $pres_1[2]$ et que l'occurrence $ETA_1[4]$ dépend de $pres_1[7]$.

Selon ce scénario, le module M_5 démarre à l'instant 0 et le module M_3 démarre avec une phase de 40 ms. Le capteur mesurant la pression démarre à 5 ms et ensuite délivre périodiquement une mesure toutes les 20 ms. Ce capteur est interconnecté avec le réseau AFDX via le *Remote Data Concentrator* RDC_1. Ce dernier démarre avec une phase de 20 ms. L'occurrence $ETA_1[2]$ est élaborée à partir de la dernière valeur de la vitesse reçue avant le démarrage de la deuxième occurrence de FM_1, c'est-à-dire $speed_1[2]$. Cette occurrence dépend à son tour de $pres_1[1]$ et donc la fraîcheur d'$ETA_1[2]$ est le temps écoulé entre la production de $pres_1[1]$ et la production de $ETA_1[2]$. Sur la figure, cette fraîcheur est $f_1 = 115 - 5 = 110$ ms. L'heure estimée d'arrivée suivante est encore élaborée à partir des mêmes occurrences de variable, ainsi la fraîcheur d'$ETA_1[3]$ est : $f_2 = 185 - 5 = 180$ ms. La mesure de pression utilisée pour déterminer $ETA_1[4]$ est $pres_1[7]$, ce qui donne une fraîcheur $f_3 = 245 - 125 = 120$ ms. Sur ce début d'exécution, l'heure estimée d'arrivée ayant la pire fraîcheur est donc $ETA_1[3]$ car elle réutilise une mesure de pression qui a « vieilli » depuis sa dernière utilisation. Pour déterminer la pire fraîcheur d'une variable, il faut donc s'intéresser à la dernière utilisation d'une variable.

Comme pour la latence, plusieurs scénarios d'exécution sont possibles en fonction de l'asynchronisme des composants du système, de la variabilité des temps d'exécution et de communication. L'exigence spécifie que, quelle que soit l'exécution et quel que soit l'affichage, la fraîcheur de $disp_1$ par rapport à $pres_1$ doit toujours être inférieure à 400 ms.

Exigence de cohérence entre chaînes fonctionnelles divergentes

Considérons la chaîne fonctionnelle liant une requête du pilote et l'affichage des informations d'un *waypoint* sur son écran $\overset{req_1}{\rightarrow} KU_1 \overset{wpId_1}{\rightarrow} FM_1 \overset{query_1}{\rightarrow} NDB \overset{answer_1}{\rightarrow} FM_1 \overset{wpInfo_1}{\rightarrow} MFD_1 \overset{disp_1}{\rightarrow}$ et la chaîne liant une requête du pilote et l'affichage des informations d'un *waypoint* sur l'écran du copilote $\overset{req_1}{\rightarrow} KU_1 \overset{wpId_1}{\rightarrow} FM_2 \overset{query_2}{\rightarrow} NDB \overset{answer_2}{\rightarrow} FM_2 \overset{wpInfo_2}{\rightarrow} MFD_2 \overset{disp_2}{\rightarrow}$. L'exigence de cohérence E_3 sur ces chaînes s'entend de la façon suivante : les premières occurrences des variables en sortie de ces chaînes fonctionnelles dépendant d'une même occurrence en entrée ne doivent jamais être distantes de plus de 500 ms. Pour illustrer cette exigence, nous étendons le scénario utilisé pour la présentation de l'exigence de latence, en ajoutant les modules M_2 et M_4 utilisés dans la deuxième chaîne fonctionnelle. Ce scénario est représenté sur la figure 1.8.

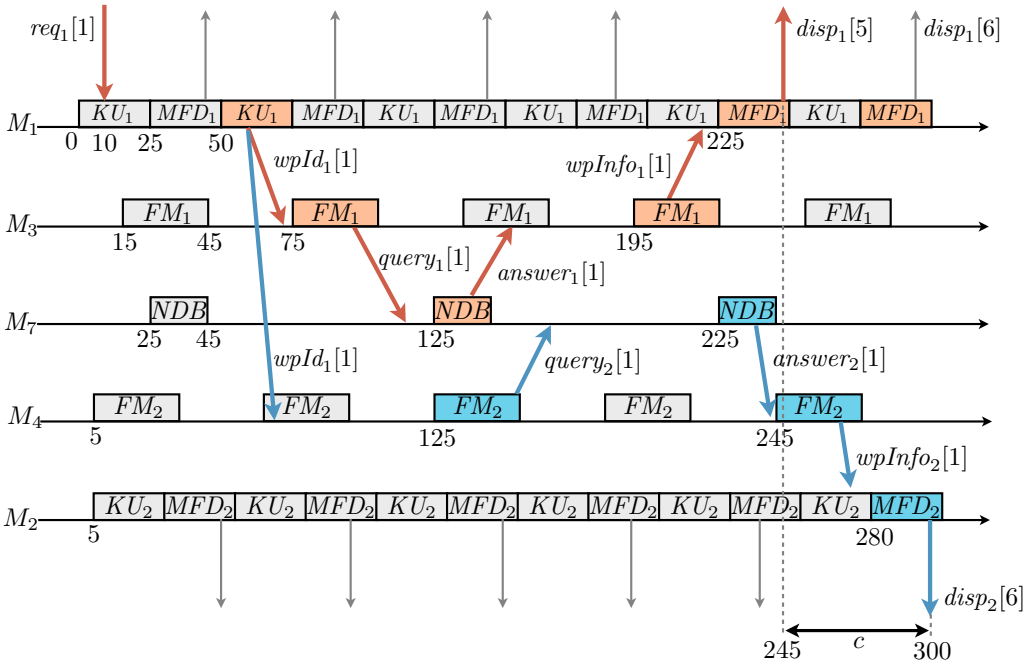


FIGURE 1.8 – Cohérence des affichages

Dans ce scénario, les premiers affichages produits par MFD_1 et MFD_2 sont cohérents dans la mesure où ils ne dépendent encore d'aucune requête du pilote. Le premier affichage dépendant d'une requête est $disp_1[5]$ qui est produit par MFD_1 à 245 ms. La mesure de la cohérence débute à cet instant et termine lorsque MFD_2 produit un affichage dépendant de $req_1[1]$. Sur cet exemple, il s'agit de $disp_2[6]$ produit à 300 ms. La valeur de la cohérence est alors $c = 300 - 245 = 55$ ms. L'occurrence $disp_1[6]$ dépend de la même requête que $disp_1[5]$, mais elle n'influence pas la cohérence. L'exigence spécifie que, quelle que soit l'exécution et quelle que soit la requête, la cohérence entre une requête à ses affichages doit être inférieure à 500 ms.

Exigence de cohérence entre chaînes fonctionnelles convergentes

Une autre exigence de cohérence est définie sur ce système, permettant d'illustrer une seconde notion de cohérence. La cohérence précédente concerne des chaînes divergentes : une entrée du système génère plusieurs sorties qui doivent respecter une contrainte temporelle. Cette deuxième exigence concerne des chaînes convergentes : une sortie du système dépend de plusieurs entrées qui doivent respecter une contrainte temporelle.

L'affichage de l'heure estimée d'arrivée à destination du pilote $disp_1$ est élaboré à partir des mesures de pression $pres_1$ et $pres_2$ réalisées respectivement par les capteurs $sensor_1$ et $sensor_2$. Les chaînes fonctionnelles impliquées sont $\overset{pres_1}{\rightarrow} ADIRU_1 \overset{speed_1}{\rightarrow} FM_1 \overset{ETA_1}{\rightarrow} MFD_1 \overset{disp_1}{\rightarrow}$ et $\overset{pres_2}{\rightarrow} ADIRU_2 \overset{speed_2}{\rightarrow} FM_1 \overset{ETA_1}{\rightarrow} MFD_1 \overset{disp_1}{\rightarrow}$. L'exigence de cohérence E_4 sur ces chaînes s'entend de la façon suivante : chaque affichage doit dépendre de mesures de pression réalisées dans un même intervalle de temps. Pour illustrer cette exigence, nous

étendons le scénario utilisé pour la présentation de l'exigence de fraîcheur, en ajoutant le module M_6 qui héberge la deuxième centrale inertielle et RDC_2 qui sert de passerelle au deuxième capteur. Ce scénario est représenté sur la figure 1.9.

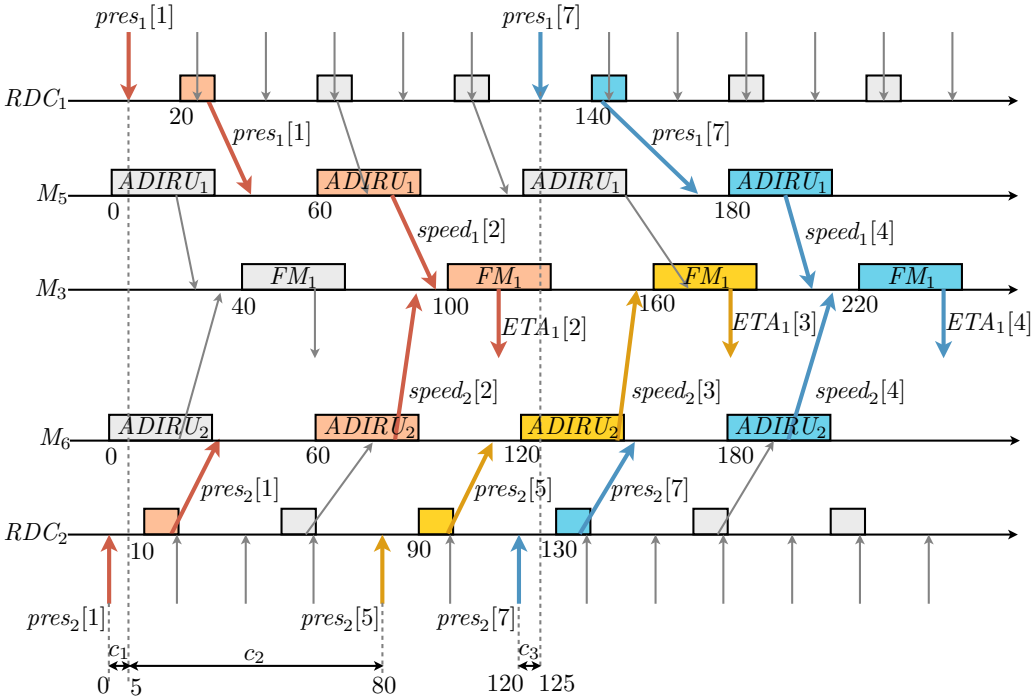


FIGURE 1.9 – Cohérence des mesures de pression

Dans ce scénario, nous identifions les dépendances entre les heures estimées d'arrivée produites par FM_1 et les occurrences des variables de pression $pres_1$ et $pres_2$. L'occurrence $ETA_1[2]$ dépend de $pres_1[1]$ mesurée à 5 ms et de $pres_2[1]$ mesurée à 0 ms. Ainsi pour cette occurrence, la cohérence est $c_1 = 5 - 0 = 5$ ms. Pour l'heure estimée d'arrivée suivante, $pres_1$ est encore utilisée mais une valeur plus récente de la pression venant du deuxième capteur est utilisé, ainsi la cohérence est moins bonne et elle vaut $c_2 = 80 - 5 = 75$ ms. L'occurrence $ETA_1[3]$ dépend de $pres_1[7]$ mesurée à 125 ms et de $pres_2[7]$ mesurée à 120 ms, la cohérence est alors $c_3 = 125 - 120 = 5$ ms. Pour vérifier l'exigence, il faut alors vérifier que quelle que soit l'exécution et quelle que soit l'occurrence de l'heure estimée d'arrivée, les occurrences des pressions dont elle dépend sont séparées d'au plus 300 ms.

1.3 Conclusion

Dans ce chapitre, nous avons présenté une étude de cas basée sur un système de gestion de vol. Ce système est décrit comme un ensemble de fonctions avioniques déployé sur une architecture IMA qui est partagée par d'autres fonctions non représentées. L'originalité de cette étude de cas provient des exigences que ce système doit satisfaire. En effet, bien que la notion de latence (ou de temps de réponse) soit une notion classique dans la littérature, les notions de fraîcheur et surtout de cohérence sont plus rarement étudiées. Ces notions ainsi que le fonctionnement du système ont été introduits à l'aide d'exemples de scénarios d'exécution.

Après analyse de cette étude de cas, nous identifions différents points nécessaires à la réalisation des objectifs de vérification. Premièrement, nous avons besoin d'une description formelle de l'architecture fonctionnelle et matérielle du système, permettant de définir les ressources d'exécution, les capteurs et actionneurs et les ressources de communication ainsi que l'allocation de ces ressources aux fonctions. Ensuite, il faut définir la sémantique du système en prenant en compte son comportement temporel mais également les dépendances entre les occurrences des variables. Finalement, la méthode de vérification proposée devra être suffisamment souple pour permettre la vérification d'exigences temps réel diverses, et elle devra également pouvoir s'appliquer à des systèmes constitués d'un nombre important de composants

asynchrones. Dans le chapitre suivant, nous dressons un état de l'art en lien avec le problème posé.

Chapitre 2

Méthodes de modélisation et de véri cation

Dans ce chapitre nous présentons un état de l'art focalisé sur les besoins identifiés lors de la présentation de l'étude de cas au chapitre précédent. En particulier, il nous semble pertinent de disposer d'un modèle de description de la structure d'un système IMA. Ainsi, dans la section 2.1 nous considérons l'utilisation de langages de description d'architecture. Au travers de la modélisation d'une partie de l'étude de cas, nous observerons qu'un langage tel qu'AADL correspond au bon niveau de description pour le problème étudié. Cependant, l'objectif de cette thèse étant l'analyse d'exigences temporelles, nous n'utilisons pas ce langage, mais plutôt un formalisme minimal comme point d'entrée des calculs et des outils.

Ensuite, pour pouvoir exprimer et vérifier les exigences temps réel du système, il faut également modéliser formellement les comportements. Pour cela, en AADL, on pourrait par exemple utiliser l'annexe comportementale [94]. Dans la section 2.2, nous mettons en œuvre deux formalismes, parmi ceux disponibles dans la littérature, adaptés à la modélisation de systèmes IMA : les *automates temporisés* et le *tagged signal model*. En premier lieu, nous avons considéré l'utilisation d'*automates temporisés* pour la modélisation. Leur expressivité permet de quantifier des durées et l'écoulement du temps en général, ce qui correspond à notre besoin. En revanche, la complexité des méthodes de vérification associées à ces modèles limite leur utilisation à des exemples restreints. C'est la raison pour laquelle nous avons alors envisagé une modélisation dans le *tagged signal model*. Il s'agit du formalisme sur lequel est fondée cette thèse. L'idée suivie est l'expression du comportement du système à l'aide de contraintes sur ses événements. Le modèle peut alors être analysé à l'aide d'outils de propagation de contraintes ou de programmation linéaire mixte.

Du fait de leur complexité, les réseaux de communication sont généralement abstraits à l'aide de leurs paramètres de qualité de service tels que leurs meilleur et pire temps de traversée. Nous dédions la section 2.3 de l'état de l'art aux techniques d'évaluation de performances, applicables aux réseaux AFDX. Nous détaillons les deux techniques qui sont aujourd'hui les plus adaptées : le *Network Calculus* et l'*approche par trajectoire*.

La section 2.4 traite de différentes méthodes destinées aux systèmes distribués temps réel et qui prennent en compte des exigences de bout en bout. En particulier, des travaux sur la théorie de l'ordonnancement s'intéressent à l'évaluation de latences de bout en bout d'un système. Le *Real-time Calculus*, qui est une généralisation du *Network Calculus*, peut également être utilisé à ces fins. Cependant, nous cherchons à vérifier d'autres exigences que des exigences de latence, nous nous sommes ainsi penchés sur les travaux dédiés aux notions de cohérence et de fraîcheur.

A noter que nous nous intéressons en général aux méthodes dites déterministes, c'est-à-dire permettant de garantir qu'une exigence est satisfaite dans tous les scénarios d'exécutions possibles. Nous ne considérons donc ni les méthodes stochastiques, ni les méthodes de simulation. Par exemple, la suite logicielle CADP (pour *Construction and Analysis of Distributed Processes*) offre la possibilité de combiner la vérification fonctionnelle et l'analyse de performance des systèmes distribués au sein d'un même outil [50]. Ces méthodes ont été appliquées avec succès dans le monde avionique [51], en particulier pour prouver la correction de protocoles de communication. En revanche, l'analyse de propriétés temporelles, telles que la latence, s'effectue à l'aide de simulations ou de modèles stochastiques. Ces analyses ne sont donc pas exhaustives et ne peuvent par conséquent pas être utilisées pour vérifier des exigences temps réel.

2.1 Modélisation d'architecture

De nombreux langages de modélisation ont été proposés dans le but d'aider à la conception des systèmes embarqués et d'automatiser une partie des étapes de développement (par exemple en intégrant de la génération automatique de code [32]). Parmi les langages les plus matures pour les applications temps réel, nous pouvons citer MARTE (*Modeling and Analysis of Real-time and Embedded systems*) [53], SysML (*System Modeling Language*) [54] et AADL (*Architecture and Analysis Design Language*) [46]. Chaque langage est focalisé sur différents niveaux d'abstraction des systèmes : par exemple SysML et MARTE sont utilisés pour représenter une vision globale du système, alors qu'AADL est plus adapté pour la description des détails d'implémentation des systèmes et de leur architecture physique. Des travaux récents proposent de combiner les avantages des différents langages. Dans [19] les auteurs définissent une extension de SysML permettant d'intégrer l'ensemble des détails d'implémentation définissables avec AADL, et dans [45] les auteurs décrivent comment modéliser les principes d'AADL dans MARTE. Dans la suite, nous nous intéressons à AADL dans la mesure où il existe des travaux (par exemple [43]) pour modéliser et analyser des architectures fondées sur la norme ARINC 653. En particulier, une annexe spécifique à AADL est en cours de standardisation pour prendre en compte la norme ARINC 653.

2.1.1 Le langage de description d'architecture AADL

Le langage de description d'architecture AADL a été introduit pour permettre la modélisation et l'analyse formelle de systèmes embarqués temps réel. Dans ce langage, un système est décrit comme un assemblage de composants qui peuvent être de nature logicielle, matérielle ou composite (un composant peut être lui-même décrit comme un assemblage de composants). La figure 2.1, tirée de [46], résume la syntaxe graphique des différents composants définis dans AADL V1.

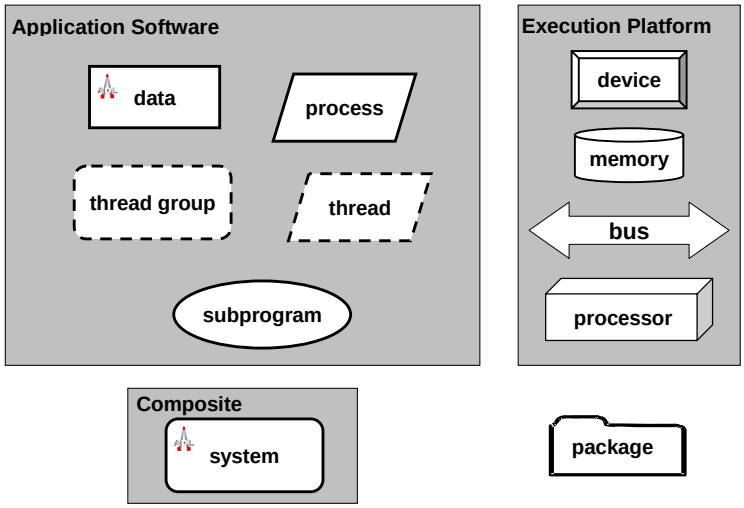


FIGURE 2.1 – Composants définis dans AADL V1

Nous détaillons les différents composants logiciels définis dans le langage, en précisant pour chacun d'eux son utilisation dans le cadre de la modélisation d'un système IMA :

data : il s'agit d'une structure de données.
Chaque variable échangée entre les fonctions est modélisée à l'aide d'une structure de données de type *data*.

thread : un *thread* sert à modéliser une tâche.
Nous ne détaillons pas l'ensemble des tâches réalisant une fonction. Un unique *thread* est utilisé pour modéliser une fonction. Les informations telles que la période et la durée d'exécution de la fonction sont contenues dans ce *thread*.

thread group : modélise la notion de hiérarchie entre différents *threads*.

Comme une fonction est modélisée par un unique *thread* et que les fonctions s'exécutent dans des partitions différentes, ce composant n'est pas utilisé dans la modélisation.

process : un *process* modélise un espace mémoire réservé à un ensemble de *threads*. Ceci implique qu'un *process* doit comporter au minimum un *thread*.
Un *process* représente la partie logicielle d'une partition. Le *thread* contenu dans un *process* modélise alors la fonction allouée à la partition.

subprogram : pour faire le lien entre un modèle AADL et du code source (fonction C, méthode Java,...), il est possible de déclarer un composant *subprogram* à l'intérieur d'un *thread*. Ce composant représente alors un programme exécutable séquentiellement (fonction C, méthode Java,...).
Ce composant n'est pas utilisé dans notre modèle.

Un composant matériel peut appartenir à l'une des catégories suivantes :

processor : un *processor* représente une abstraction du logiciel et du matériel chargés de l'exécution et de l'ordonnancement des *threads*.
un *processor* modélise le CPU d'un module IMA.

virtual processor (AADL V2) : un *processor* peut contenir plusieurs *virtual processors* ce qui permet d'ordonnancer des groupes de *threads* indépendamment les uns des autres sur un même *processor*.
un *virtual processor* modélise une partition.

device : il s'agit d'un équipement matériel interagissant avec l'environnement extérieur et dont nous ne voulons ou ne pouvons pas détailler le fonctionnement interne.
une *device* modélise un capteur ou un actionneur.

bus : il s'agit d'une abstraction du matériel permettant l'échange de données entre différents équipements.
les capteurs, les actionneurs et les modules sont reliés par des *buses*. Nous utilisons également un *bus* pour abstraire l'ensemble du réseau AFDX.

virtual bus (AADL V2) : un *bus* peut contenir plusieurs *virtual buses*, ce qui permet de modéliser plusieurs canaux, avec par exemple une qualité de service différenciée, au sein d'un même bus.
nous utilisons des *virtual buses* pour différencier les VL au sein du *bus* qui abstrait le réseau AFDX.

memory : il s'agit du matériel permettant de stocker du code ou des données.
le composant *memory* n'est pas utilisé dans le modèle.

Différents types d'interaction sont définissables entre composants. Les interfaces d'un composant sont appelées ses *features*. Pour modéliser les échanges inter-partitions tels que définis dans la norme ARINC 653, nous utilisons les *features* suivants :

data port : pour un port en *sampling*,
event data port : pour un port en *queuing*,

Les besoins de connexions entre composants matériels sont définis par les *features* :

requires bus access : pour préciser que l'équipement nécessite une connexion à un certain *bus*,
provides bus access : pour préciser que l'équipement fournit une connexion à un certain *bus*.

Maintenant que les différents composants AADL sont définis, nous les utilisons pour modéliser une partie de l'étude de cas.

Modélisation d une partie de l étude de cas

Pour évaluer les avantages et les inconvénients de ce langage, nous modélisons en partie le système de gestion de vol (FMS) de l'étude de cas. L'outillage de modélisation et d'analyse disponible (OSATE 1.5.8) ne permet pas de prendre en compte la version 2 d'AADL. En particulier, la notion de *virtual processor* nécessaire pour définir les partitions ARINC 653 n'est pas disponible. Nous ne pouvons donc pas modéliser intégralement l'étude de cas avec l'outillage actuel. Nous détaillons tout de même les

principes à appliquer.

La sous-partie du FMS que nous modélisons est composée de :

- *keyboard* (le clavier) : ce capteur transmet la requête du pilote (`request_pilot`) au gestionnaire du clavier *KCCU*,
- *KCCU* : transmet un identifiant de point de navigation (`waypoint_identifier`) au gestionnaire de vol *FM*,
- *FM* : transmet les informations du *waypoint* (`waypoint_information`) au gestionnaire d'affichage *MFD*,
- *MFD* : transmet la page à afficher (`display`) à l'écran *display*
- *display* (l'écran) : cet actionneur affiche la page avec les informations du *waypoint*.

La modélisation se construit avec une approche « *bottom-up* » : nous déclarons en premier lieu les *data*, puis les *threads* qui les utilisent, puis les *processes*... jusqu'au système complet. Nous commençons donc par la déclaration des données utilisées dans le système. A titre d'exemple, la déclaration d'un identifiant de point de navigation se fait comme suit :

```
data waypoint_identifier
properties
   Source_Data_Size => 600 bits;
end waypoint_identifier;
```

Notons que le langage permet d'attacher des propriétés aux composants. Dans cet exemple, nous précisons la taille de la variable de la donnée `waypoint_identifier` en attribuant la valeur `600 bits` à la propriété `Source_Data_Size`.

Cette variable est produite par la fonction *KCCU* gérant les entrées du clavier. Nous définissons les composants logiciels de cette fonction et de sa partition. Nous commençons par le *thread* de la fonction :

```
thread KCCU
features
   req : in event data port request_pilot;
   wpId : out event data port waypoint_identifier;
properties
   Dispatch_Protocol => Periodic;
   Period => 50 ms;
   Compute_Execution_Time => 0ms .. 25ms;
flows
   flowKCCU : flow path req -> wpId ;
end KCCU;
```

L'entrée du *thread* est un port en *queuing* qui stocke des variables de type `request_pilot`. A la fin de son exécution, le *thread* envoie une variable de type `waypoint_identifier` dans un port également en *queuing*. Les propriétés utilisées pour ce *thread* précisent que la fonction est périodique, avec une période de 50 ms et un temps d'exécution compris dans l'intervalle $[0, 25]$ (en ms). Nous utilisons la notion de flux (mot clef **ows**) afin de définir un chemin logique au sein d'un composant. Cette notion permet d'exprimer des morceaux de chaînes fonctionnelles sur lesquelles reposent les exigences temps réel. Par exemple, `flowKCCU : flow path req -> wpId` est utilisé ici pour déclarer que dans ce *thread* le port `req` est fonctionnellement associé au port `wpId`, i.e. les variables écrites sur le port de sortie `wpId` dépendent des variables lues sur le port d'entrée `req`.

Une partition se définit à l'aide d'un composant *process*. Celui-ci est composé de deux parties. La première définit les interfaces de la partition, c'est-à-dire ses entrées et sorties. Une partition ne contient qu'une unique fonction et donc le *process* de la partition ne contient qu'un unique *thread*. C'est donc naturellement que dans cet exemple le *process* `process_KCCU` offre les mêmes interfaces que le *thread* `KCCU`.

```
process process_KCCU
features
   req : in event data port request_pilot;
   wpId : out event data port waypoint_identifier;
end process_KCCU;
```

La deuxième partie du composant est son implémentation qui contient sa structure interne, c'est-à-dire la liste de ses sous-composants, les connexions entre ses sous-composants et les propriétés spécifiques à l'implémentation.

```
process implementation process_KCCU.impl
subcomponents
   function_KCCU : thread KCCU;
connections
   portIn : req -> function_KCCU.req;
   portOut :  function_KCCU.wpId -> wpId;
flows
   flowProcKCCU : flow path
                        req -> portIn -> function_KCCU.flowKCCU ->
                        portOut -> wpId;
end process_KCCU.impl;
```

Dans notre modèle de partition, `fonction_KCCU` est un sous-composant (*subcomponent*) *thread* de type `KCCU` du *process* de la partition. Les liens entre les entrées-sorties du composant et celles des sous-composants sont réalisés par la notion de *connections*. Enfin, le chemin de données logique est défini par la notion de *ows* : l'entrée `req` et la sortie `wpId` sont reliées via la connexion `portIn`, puis le chemin `flowKCCU` (défini dans le sous-composant `function_KCCU`) et finalement la connexion `portOut`. La modélisation graphique de ce composant est représentée sur la figure 2.2.

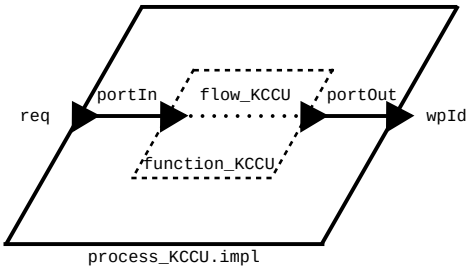


FIGURE 2.2 – Implémentation d'un *process* pour la fonction KCCU

Les partitions, décrites à l'aide des *process*, s'exécutent sur un processeur (mot clef *processor*) en respectant les contraintes temporelles de la MAF du module, c'est-à-dire les intervalles de temps d'exécution des partitions.

```
processor implementation procM1.impl
subcomponents
   partition_KCCU : virtual processor partition.impl;
   partition_MFD : virtual processor partition.impl;
properties
   ARINC653::Module_Major_Frame => 50ms;
   ARINC653::Partition_Slots => (25ms, 25ms);
   ARINC653::Slots_Allocation => ( reference (partition_KCCU),
                                   reference (partition_MFD));
end procM1.impl;
```

Dans l'exemple, nous considérons le processeur du module M_1, `procM1`. Sur ce processeur s'exécutent les partitions *KCCU* et *MFD*. Leurs contraintes temporelles sont décrites à l'aide de deux processeurs virtuels (mot clef *virtual processor*). Enfin, les exécutions sur le processeur du module doivent respecter les contraintes définies dans la norme ARINC 653, à savoir la durée de la MAF, la durée des intervalles de temps alloués aux partitions et l'ordre d'exécution des partitions dans la MAF. Ces différentes propriétés sont préfixées par `ARINC653::` pour indiquer qu'elles sont définies dans l'annexe ARINC 653 et non pas dans les propriétés fondamentales d'AADL.

Les propriétés nécessaires pour définir les partitions sont :

- `ARINC653::Module_Major_Frame => 50ms` (définit la durée de la MAF),
- `ARINC653::Partition_Slots => (25ms, 25ms)` (définit deux intervalles de temps de longueur 25 ms),
- `ARINC653::Slots_Allocation => ( reference (partition_KCCU), ...)` (alloue le premier intervalle au *virtual processor* `partition_KCCU` et le deuxième à `partition_MFD`).

Avant de pouvoir déclarer le module M_1, nous devons déclarer les moyens de communication auxquels il est connecté. Le système considéré contient trois *bus* au sens AADL : un *bus* CAN pour accéder au

clavier, un autre pour l'écran et un dernier *bus* qui sert d'abstraction au réseau AFDX. Nous ne détaillons que la définition du composant utilisé pour le réseau AFDX. AADL ne permet pas de décrire de manière simple un réseau, aucun vocabulaire dédié n'existe pour décrire les tables de routage des commutateurs par exemple. C'est la raison pour laquelle nous nous contentons dans cet exemple de modéliser l'ensemble du réseau avec un *bus* contenant des *virtual buses* représentant les VL du réseau. Nous déclarons d'abord l'implémentation du *virtual bus* utilisé pour VL_1 :

```
virtual bus implementation VL.1
properties
   Allowed_Message_Size => 512B..12144B;
   Latency => 120us..2ms;
end VL.1;

bus implementation AFDX.impl
subcomponents
   VL1: virtual bus VL.1;
   VL2: virtual bus VL.2;
end ADFX.impl
```

`VL.1` est une implémentation du *virtual bus* `VL` (non décrit ici). La propriété `Allowed_Message _Size` permet de spécifier les tailles minimale et maximale des messages attendus sur le VL, et `Latency` spécifie les temps de traversée meilleur et pire cas du VL. Dans cet exemple, nous ne définissons pas les lisseurs de trafic des VL. La latence déclarée dans le composant doit donc comprendre à la fois le temps de traversée du réseau et le temps de traversée du lisseur de trafic. Il est donc nécessaire d'effectuer l'évaluation des performances du réseau préalablement à la modélisation. L'implémentation du *bus* `AFDX.impl` est alors définie par l'ensemble des VL.

Pour représenter un module IMA, nous définissons un système composé d'un *processor* (contenant des *virtual processors*) et des *processes* pour les partitions qui s'exécutent sur le module. Nous déclarons un type pour le module M_1 (qui appartient à la catégorie des *systems* car c'est un assemblage de composants logiciels et matériels) pour préciser les interfaces logicielles et matérielles du module :

```
system moduleM1
features
   -- software interfaces
   req : in event data port request_pilot;
   wpInfo : in event data port waypoint_information;
   wpId : out event data port waypoint_identifier;
   disp : out data port display;

   -- hardware interfaces
   can_keyboard: requires bus access can.keyboard;
   can_display: requires bus access can.display;
   afdx : requires bus access afdx.impl;

end moduleM1;
```

Nous définissons alors une implémentation de ce module :

```
system implementation moduleM1.impl
subcomponents
   cpu : processor procM1.impl;
   partKCCU : process process_KCCU.impl;
   partMFD : process process_MFD.impl;
connections
   conn_req : req -> partKCCU.req;
   conn_wpInfo : wpInfo -> partMFD.wpInfo;
   conn_wpId : partKCCU.wpId -> wpId;
   conn_disp : partMFD.disp -> disp;
flows
   fl_req_wpId : flow path
                        req -> conn_req ->
                        partKCCU.flowProcKCCU -> conn_wpId;
   fl_wpInfo_disp : flow path
                        wpInfo -> conn_wpInfo ->
                        partMFD.flowProcMFD -> conn_disp;
end moduleM1.impl;
properties
actual_processor_binding =>
      (reference (cpu.partition_KCCU)) applies to partKCCU;
```

```
actual_processor_binding =>
      (reference (cpu.partition_MFD)) applies to partMFD ;
```

La *properties* telles que `actual_processor_binding =>` permettent de déclarer l'allocation d'un *process* sur un *processor* (ou un *virtual processor*). Le paramètre `(reference (cpu.partition_KCCU)) applies to partKCCU` spécifie alors que le *process* `partKCCU` est alloué sur `cpu.partition_KCCU`. La figure 2.3 représente la version graphique de l'implémentation du module M_1.

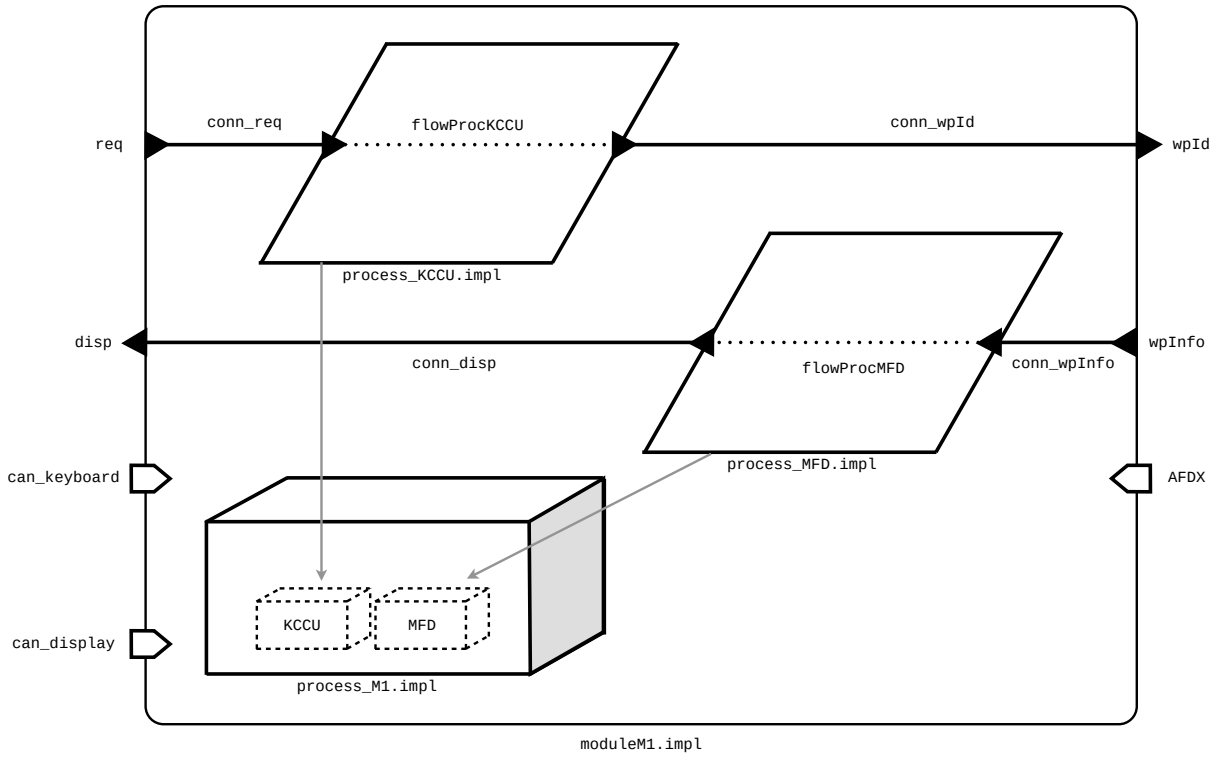


FIGURE 2.3 – Implémentation du module M_1

Les capteurs et actionneurs sont modélisés à l'aide de *devices*. Chaque *device* du système requiert un accès à un *bus* et dispose d'un port de données (produisant *req* pour le clavier et acceptant *disp* pour l'écran).

Finalement, nous déclarons le système global qui contient l'ensemble des composants du FMS :

```
system FMS
end FMS;

system implementation FMS.impl
subcomponents
   bus_can_key : bus can_keyboard;
   bus_can_disp: bus can_display;
   bus_AFDX: bus afdx.impl;

   module1: system moduleM1.impl;
   module2: system moduleM2.impl;

   dev_key: device keyboard.impl;
   dev_disp: device display.impl;
connections
   conn_req : dev_key.req -> module1.req;
   conn_wpId : module1.wpId -> module2.wpId;
   conn_wpInfo : module2.wpInfo -> module1.wpInfo;
   conn_disp : module1.disp -> dev_disp.disp;

   busAccess1 : bus access bus_can_key -> dev_key.can_keyboard;
   busAccess2 : bus access bus_can_key -> module1.can_keyboard;
```

```
    busAccess3 : bus access bus_can_disp -> dev_disp.can_display;
    busAccess4 : bus access bus_can_disp -> module1.can_display;
    busAccess5 : bus access bus_AFDX -> module1.afdx;
    busAccess5 : bus access bus_AFDX -> module1.afdx;
properties
    Actual_connection_binding =>
        (reference (bus_can_key)) applies to conn_req;
    Actual_connection_binding =>
        (reference (bus_AFDX.VL1)) applies to conn_wpId;
    Actual_connection_binding =>
        (reference (bus_AFDX.VL2)) applies to conn_wpInfo;
    Actual_connection_binding =>
        (reference (bus_can_disp)) applies to conn_disp;
flows
    eteLatency : end to end flow
       dev_key.fl_src -> conn_req -> module1.fl_req_wpId ->
       conn_wpId -> module2.fl_wpId_wpInfo ->
       conn_wpInfo -> module1.fl_wpInfo_disp ->
       conn_disp -> dev_disp.fl_sink { expected latency => 700 ms;};
end FMS.impl;
```

Les *connections* telles que `bus access bus_can_key -> dev_key.can_keyboard` permettent de déclarer les connexions entre équipements physiques (ici, entre le *bus* `bus_can_key` et l'interface du la *device* `dev_key`). Les *properties* `Actual_connection_binding` permettent d'allouer une connexion logique à un moyen de communication physique. Par exemple avec les paramètres `(reference (bus_AFDX.VL1)) applies to conn_wpId`, nous spécifions que la connexion logique `conn_wpId` est allouée sur le *virtual bus* `VL1` du *bus* `bus_AFDX`. Le *ow* `eteLatency` est déclaré comme un flux de bout en bout dans le système, nous lui attachons la propriété `expected latency => 700 ms` qui permet de spécifier la latence maximale attendue le long de ce chemin. La chaîne fonctionnelle associée à cette exigence peut alors être retrouvée en parcourant ce flux. La figure 2.4 résume le système décrit dans cet exemple.

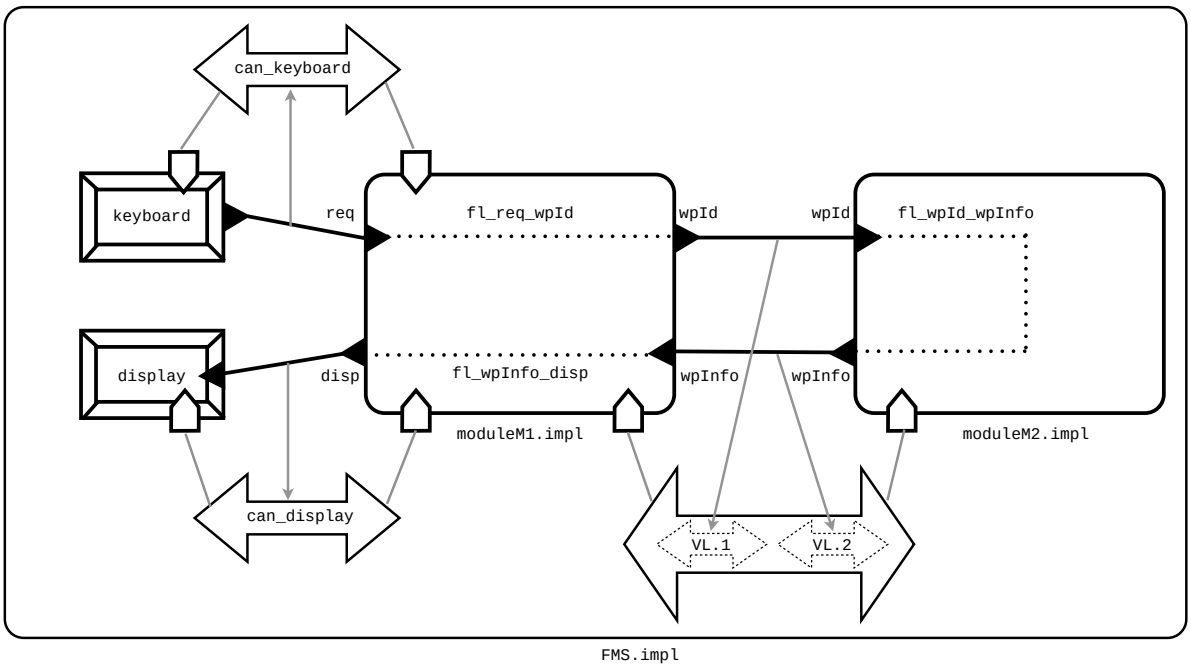


FIGURE 2.4 – Implémentation du système complet FMS

Evaluation de performance sur un modèle AADL

Dans [47] les auteurs décrivent comment utiliser la notion de *ows* pour spécifier une latence. L'évaluation de la latence pire cas est effectuée en additionnant le pire temps de réponse *local* de chaque composant traversé. Le pire temps de réponse d'un composant est déduit de ses spécifications. Cette technique est

implémentée dans le plug-in *Flow Latency Analysis* de l'outil OSATE 1.5.8. Cet outil ne prenant pas encore en compte les spécificités d'AADL V2, ni l'annexe ARINC653, nous ne pouvons donc pas l'utiliser pour analyser le système. Cependant, nous développerons au chapitre 8 une approche similaire, que nous appelons *méthode locale* et qui servira de base de comparaison avec la méthode d'évaluation plus complexe, car fondée sur la résolution de programme linéaire mixte, que nous proposons.

Dans [43] les auteurs évaluent l'ordonnançabilité des tâches des partitions d'un module à l'aide de simulations exhaustives. En particulier, les propriétés temporelles d'un module sont traduites dans un modèle exploitable par le simulateur de l'outil *Cheddar* [99], qui se charge alors de la validation de l'ordonnançabilité. A noter que cette approche ne prend pas en compte l'aspect distribué du système, dans la mesure où elle ne considère pas les communications entre modules, mais uniquement les communications à l'intérieur d'un même module.

Nous trouvons également dans la littérature des travaux (par exemple [100]) utilisant le *Real-Time Calculus* [104] pour évaluer des latences sur un modèle AADL. Nous présenterons le *Real-Time Calculus* à la sous-section 2.4.1.

Une autre approche possible pour l'analyse d'un modèle AADL est l'utilisation du *model cheking*. Par exemple dans [93] l'auteur définit un « mini AADL » se concentrant sur le système étudié (logiciel de vol embarqué dans un satellite). Ce « mini AADL » ne prend pas en compte l'architecture matérielle du système et restreint les composants logiciels au strict nécessaire pour la modélisation du système considéré. Ensuite, la sémantique du comportement du modèle est formalisée en TLA+ (pour *Temporal Logic of Action*) [65], ce qui permet l'utilisation d'un *model checker* (en l'occurrence TLC) pour vérifier des propriétés temporelles sur ce modèle et en particulier des propriétés d'ordonnançabilité. Outre le fait que cette approche ne permet pas de décrire un système IMA, la vérification d'exigences par *model checking* limite la dimension des systèmes analysés.

Conclusion

Le niveau de description d'un langage tel qu'AADL correspond donc bien à notre besoin, malgré le fait qu'il manque un « vocabulaire » pratique permettant de décrire un réseau complexe comme l'AFDX. Dans cette thèse, l'accent est mis sur l'évaluation de performance, nous faisons donc le choix de décrire les éléments du système à l'aide d'un formalisme plus simple, dans le sens où il est dédié à notre problème. Ce formalisme peut être vu comme une représentation intermédiaire non visible par l'utilisateur. Il sera présenté au chapitre 4. Nous conserverons d'AADL la décomposition d'un système suivant différentes vues : architecture fonctionnelle, architecture matérielle et allocation. La problématique est alors d'exprimer formellement le comportement du système.

2.2 Modélisation de comportement

Dans cette section, nous nous intéressons aux formalismes permettant d'exprimer la sémantique temporelle d'un système, c'est-à-dire qui définissent formellement les comportements possibles du système. Nous nous intéressons à une représentation du temps *physique* dans le sens où les exigences que nous devons vérifier s'expriment sur des durées entre des instants. Il est nécessaire que le formalisme choisi permette de mesurer explicitement l'écoulement du temps. Nous présentons les *automates temporisés* et le *tagged signal model* qui sont deux formalismes utilisables de cette façon. Le *tagged signal model* est le formalisme qui a été retenu dans cette thèse. Nous présentons également brièvement le langage CCSL (pour *Clock Constraint Specification Language*) pour son lien avec le *tagged signal model*. En effet, ce langage a été introduit pour spécifier et analyser les comportements de modèle MARTE et le modèle de temps de MARTE a pour inspiration le *tagged signal model*.

2.2.1 Approche avec automates temporisés et model checking

Les automates temporisés ont été introduits par Alur et Dill [8, 7] pour décrire le comportement de systèmes en prenant en compte leur aspect temporel. La vérification de modèles, décrits dans ce formalisme, est possible grâce à l'association de techniques de vérification tels que le *model checking* [38, 22].

Automates temporisés

Un automate temporisé est un automate fini, étendu avec un ensemble d'horloges et des règles supplémentaires sur les transitions. Les horloges sont des variables positives réelles qui évoluent à la même vitesse dans le temps. Les transitions d'un tel automate peuvent être étiquetées par :

- une garde portant sur les valeurs des horloges, c'est-à-dire une condition sur les horloges indiquant si la transition peut être franchie,
- une action effectuée lors du tir de la transition,
- des remises à zéro d'horloges effectuées lors du tir de la transition.

Formellement, un automate temporisé $\mathcal{A}$ est un 6-uplet $(L, l_0, X, \Sigma, \quad, Inv)$ avec :

1. L un ensemble fini de localités,
2. l_0 la localité initiale,
3. X un ensemble fini d'horloges, dont les valeurs sont dans $\mathbb{R}^+$,
4. Σ un ensemble fini d'actions,
5. $\subseteq L \times \mathcal{C}(X) \times \Sigma \times 2^X \times L$ un ensemble fini de transitions. Une transition $e = \ l, \gamma, \sigma, R, l$ correspond à une transition entre la localité l et la localité l , gardée par la contrainte d'horloge γ, étiquetée par l'action σ et qui remet les horloges de $R \subset X$ à zéro,
6. $Inv : L \quad \mathcal{C}(X)$ est une fonction qui associe des invariants aux localités. Les invariants d'une localité sont des contraintes sur les horloges de la forme $Inv(l) = \bigwedge x \leq p$ où x est une horloge et $p \quad \mathbb{N}$.

La sémantique d'un automate est donnée par un système de transitions dans lequel un état consiste en la localité courante et une valuation des horloges. Nous ne détaillons pas cette sémantique mais il y a deux types de transition entre états :

- l'automate peut laisser passer le temps dans une localité tant que son invariant est satisfait,
- l'automate peut tirer une transition si la garde de la transition est satisfaite et que l'invariant de la localité atteinte l'est également.

Cette sémantique implique que les transitions de l'automate sont instantanées et le temps ne peut s'écouler que dans les localités.

Exemple 1 (Un automate temporisé)**.** *Nous illustrons la notion d'automate temporisé avec un exemple tiré de [21] et représenté sur la figure 2.5.*

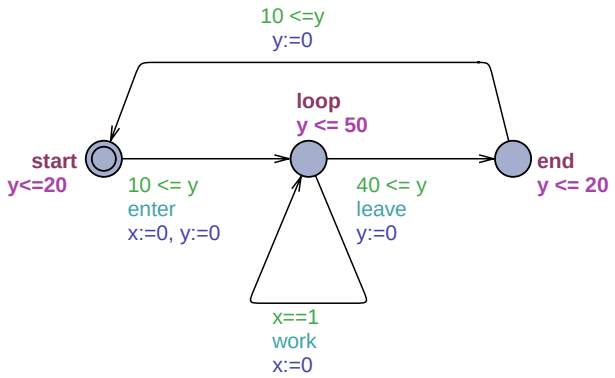


FIGURE 2.5 – Exemple d'automate temporisé

*Cet automate comporte trois localités (*start *qui est la localité initiale,* loop *et* end*) et son comportement temporel est contrôlé par deux horloges* x *et* y*. Un invariant est défini pour chaque localité. Par exemple l'invariant* $y \leq 20$ *est associé à la localité* start*, ce qui signifie que l'automate peut être dans cette localité uniquement si l'horloge* y *vaut moins que 20. Trois actions sont définies (*enter*,* work *et* leave*) sur les transitions. Aucune action n'est associée à la transition entre* end *et* start*. L'horloge* x *est utilisée pour contrôler la boucle sur la localité* loop*. Cette transition ne peut être tirée que si* $x = 1$ *et cette horloge est remise à zéro quand cette transition est tirée. Ceci permet de limiter les répétitions de l'action* work*. L'horloge* y *contrôle l'exécution de l'automate complet. L'automate peut quitter la localité* start *à n'importe quel instant quand* y *est dans l'intervalle entre 10 et 20 (l'invariant de la localité* start *oblige l'automate à quitter cette localité au plus tard à 20). Il peut aller de* loop *vers* end *quand* y $[40, 50]$*, et finalement il peut aller de* end *vers* start *quand* y $[10, 20]$.

Véri cation par model checking

Les automates temporisés permettent la définition formelle du comportement d'un système. Un tel modèle peut alors être exploité pour la vérification d'exigences, en particulier à l'aide d'outils de *model checking* [38, 22]. L'idée du *model checking* est de parcourir l'ensemble des états atteignables d'un modèle pour vérifier qu'une exigence est satisfaite. Avec les automates temporisés, ceci pose deux problèmes. En premier lieu, l'ensemble des états du système n'est pas fini car les horloges ne sont pas nécessairement bornées et car le temps est dense. Pour dépasser ce problème, des techniques basées sur des relations d'équivalence ont été mises au point dans [25]. L'idée est de définir des *régions* à l'intérieur desquelles toute valuation des horloges conduit nécessairement à un même état du système. Ces équivalences permettent de définir une représentation finie des états d'un automate temporisé. La seconde difficulté concerne l'explosion combinatoire du *model checking*. Du fait du parcours exhaustif des états du modèle, la complexité de la méthode est exponentielle : bien que le nombre de *régions* d'un automate soit fini, ce nombre croît de manière exponentielle avec le nombre d'horloges du modèle.

L'expression d'une exigence peut se faire de deux manières. La première consiste à l'encoder dans le langage d'une logique. Le choix de cette logique se fait en rapport avec l'exigence à modéliser. Par exemple, la logique CTL (pour *Computational Tree Logic*) permet d'exprimer une propriété du type « l'événement a se produira toujours après l'événement b » tandis qu'en logique TCTL (pour *Timed Computational Tree Logic*) il est possible de quantifier temporellement une propriété, par exemple « l'événement a se produira toujours 5 unités de temps après l'événement b ». Pour vérifier les propriétés exprimées à l'aide de ces logiques, il existe différents outils parmi lesquels KRONOS [111], Uppaal [67, 20]... Le rapport [23] propose une comparaison des outils courants de *model checking*. Une deuxième approche consiste à exprimer l'exigence à l'aide d'un automate de test. Comme présenté dans [33, 1, 22], un automate de test implémente un état dit *unhappy* qui est atteint si l'exigence n'est pas satisfaite. Ainsi, la vérification devient un problème d'atteignabilité : si l'état *unhappy* est atteignable alors l'exigence n'est pas satisfaite. Par rapport à l'expression d'exigences avec une logique dédiée, cette approche offre de meilleures performances, mais au prix d'une expressivité plus limitée.

Modélisation de l étude de cas

Nous avons proposé une modélisation d'un système IMA à l'aide d'automates temporisés dans l'article [72]. Mais la modélisation détaillée du réseau limite la vérification d'exigence à des systèmes de taille réduite. Une solution pour dépasser cette limitation est de considérer le réseau comme un ensemble de canaux temporisés [34, 35]. Chaque canal représente alors un lien entre un unique émetteur et un unique récepteur, et ce lien impose un délai à chaque message échangé entre l'émetteur et le récepteur. Pour chaque message, le délai est choisi de façon non-déterministe dans un intervalle, borné par les meilleur et pire temps de traversée du réseau. Ces bornes sont calculables à l'aide de techniques telles que le *Network Calculus* ou l'*approche par trajectoire* que nous décrivons ultérieurement (c.f. section 2.3 page 34).

Dans la suite, nous suivons l'idée de cet article pour modéliser une partie de l'étude de cas et ainsi évaluer l'efficacité de cette approche. Nous utilisons les extensions des automates temporisés qui sont définies dans le *model checker* Uppaal. En particulier, il est possible de définir des variables (de type entier par exemple). La mise à jour d'une variable s'effectue lors du tir d'une transition : une règle d'affectation de valeurs (`u = v + 1` par exemple) ou une fonction encodée dans le langage C est alors associée à une transition.

Un des intérêts des automates temporisés est qu'un système peut être modélisé comme un réseau d'automates, c'est-à-dire la composition d'un ensemble d'automates élémentaires. La composition d'automates est définie par le *produit synchronisé*. La synchronisation de deux automates s'effectue à l'aide d'une action commune a. Cette action est suffixée par ! dans un des automates et par ? dans l'autre : l'action $a!$ exécutée par un premier automate correspond à une action $a?$ exécutée en parallèle dans un second automate. Autrement dit, une transition qui exécute l'action $a!$ ne peut être tirée que si la transition exécutant $a?$ peut l'être également. Dans la syntaxe dUppaal, une telle action est appelée un canal (*channel*).

Le système est donc modélisé comme un réseau d'automates temporisés étendus. La figure 2.6 présente ce réseau.

Nous utilisons un automate pour :
- le module M_1 (sur lequel s'exécutent les fonctions *KCCU* et *MFD*),
- le module M_2 (avec la fonction *FM*),
- le clavier (*keyboard*),

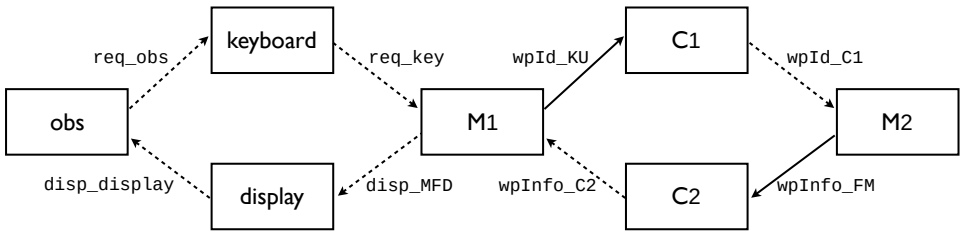


FIGURE 2.6 – Réseau d'automates temporisés du système FMS

- l'écran (*display*),
- le canal temporisé reliant *KCCU* à *FM* (noté C_1),
- le canal temporisé reliant *FM* à *MFD* (noté C_2),
- l'observateur encodant l'exigence à vérifier

Les flèches entre les automates représentent les variables utilisées. Une flèche en trait plein précise qu'il existe en plus un canal de synchronisation entre deux automates.

Le rôle de l'observateur est d'initier une requête et de mesurer le temps écoulé jusqu'à l'affichage sur l'écran d'une page dépendant de cette requête. L'observateur contrôle que cette latence ne dépasse pas un certain seuil. L'automate correspondant est représenté sur la figure 2.7. L'observateur ne peut atteindre l'état *unhappy* que si il s'est écoulé un temps supérieur à `maxLatency` depuis son démarrage et qu'aucun affichage dépendant de sa requête n'a été produit (ce qui est testé par `disp_display == 0`).

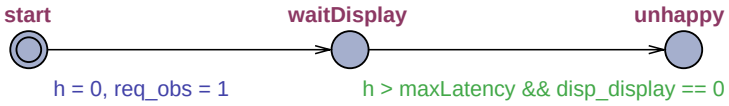


FIGURE 2.7 – Automate temporisé de l'observateur

Pour identifier qu'un affichage dépend bien de la requête initiée par l'observateur, nous utilisons des variables globales. Une variable indique si la requête de l'observateur a déjà été traitée par un élément du système. Initialement, toutes ces variables valent 0. Par exemple, la variable `req_obs` modélise l'envoi d'une requête par l'observateur : elle passe à 1 lorsque l'observateur démarre. Cette variable est utilisée dans l'automate modélisant le clavier, représenté sur la figure 2.8.

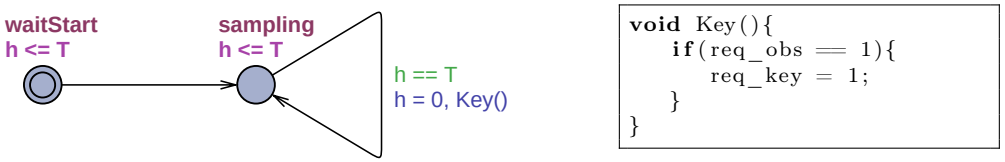


FIGURE 2.8 – Automate temporisé du clavier

Après son démarrage, l'automate du clavier boucle sur la localité *sampling*, dont la transition est tirée périodiquement (avec une période `T`). Le franchissement de cette transition modélise l'échantillonnage de l'entrée du clavier. A chaque transition, la fonction `Key()` est appelée. Elle permet de propager la valeur de `req_obs` vers `req_key`. Si `req_key` vaut 1, cela signifie qu'une requête dépendant de la requête de l'observateur est disponible en sortie du clavier. Cette valeur est exploitée dans l'automate décrit sur la figure 2.9 qui modélise le module M_1 sur lequel s'exécute les fonctions *KCCU* et *MFD*.

En accord avec le standard ARINC 653, les partitions d'un module sont ordonnancées au sein d'une trame temporelle périodique appelée la MAF (pour *MAjor time Frame*). L'automate modélise cet ordonnancement et on y retrouve les paramètres utilisés dans le modèle AADL, présenté précédemment. La durée de la MAF est modélisée par la contrainte `h <= HP`, `HP` correspond donc au paramètre AADL `ARINC653::Module_Major_Frame` et est égale à 50 ms. Les paramètres `oKU` et `oMFD` correspondent aux décalages (*o* pour *o set*) des démarrages des partitions *KCCU* et *MFD* par rapport au démarrage de la

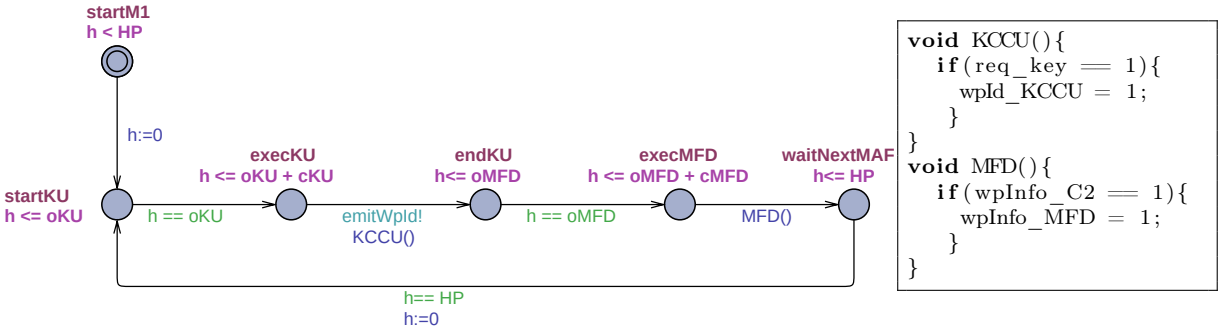


FIGURE 2.9 – Automate temporisé modélisant la MAF du module M_1

MAF. Les *slots* de temps de la MAF sont définis par `ARINC653::Partition_Slots => (25ms, 25ms)` et *KCCU* est alloué au premier slot et *MFD* au second. On a alors `oKU = 0` et `oMFD = 25`. Les paramètres `cKU` et `cMFD` sont les durées des partitions et donc des *slots*, ils valent donc 25 ms. La mise à jour de la sortie de *KCCU* (et donc la propagation de `req_key` vers `wpId_KCCU`) est modélisée par la fonction `KCCU()`. Cet automate est synchronisé avec l'automate du canal temporisé entre M_1 et M_2 (C_1) avec le canal `emitWpId`. Cette synchronisation permet de modéliser le délai subi par un message émis par *KCCU* dans le canal (l'horloge chronométrant ce délai est remise à zéro lors de la synchronisation). La figure 2.10 représente l'automate du canal temporisé C_1.

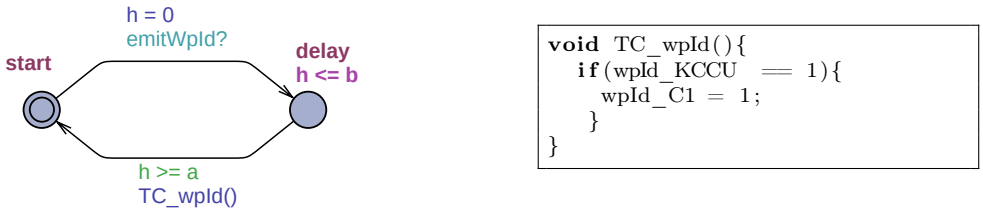


FIGURE 2.10 – Automate temporisé modélisant un canal temporisé

Les paramètres de cet automate sont les temps minimal a et maximal b de traversée du canal. Ils sont définis dans le *virtual bus* `VL.1` du modèle AADL avec la propriété `Latency => 120 us .. 2 ms`.

Les autres éléments de l'exemple sont modélisés de façon similaire. Nous utilisons ce modèle pour évaluer les capacités du *model checker*. Une particularité du système étudié est le rapport de temps entre les paramètres des fonctions qui sont définies en milli-secondes et les paramètres du réseau qui sont en micro-secondes. Par exemple, la période de *KCCU* vaut 50 ms et la valeur maximale de la latence acceptable est 700 ms. Mais le temps d'émission d'une trame réseau de taille maximale est 120 μs. Ceci pose un problème au *model checker*. En effet, cette disparité des horloges (certaines peuvent être remises à jour beaucoup plus fréquemment que d'autres), fait que le graphe des *régions* utilisé pour construire une représentation finie des états d'un modèle « explose ».

Pour vérifier que la latence de cet exemple ne dépasse pas la borne `maxLatency`, nous demandons au *model checker* de vérifier que le modèle n'atteint jamais l'état *unhappy* de l'observateur. L'outil UPPAAL produit une réponse positive : l'état n'est jamais atteint et donc la propriété est satisfaite. Cette vérification requiert un peu plus de 42 minutes [1], ce qui correspond à l'exploration de 2263053 états.

Conclusion

A la suite de ces expérimentations, nous avons noté plusieurs freins à l'utilisation de ces méthodes. En premier lieu, la modélisation d'un système IMA, asynchrone par nature, nécessite l'utilisation de nombreuses horloges. Or, la complexité de la résolution du problème augmente de façon exponentielle avec le nombre d'horloges. En outre, le système fonctionne avec plusieurs échelles de temps (période d'une fonction en ms, temps d'émission d'une trame en μs) ce qui limite l'efficacité des techniques de réduction de l'espace d'états du système.

1. vérification effectuée avec UPPAAL 4.0.11 sous MAC OS 10.6.8 et un processeur Intel Core 2 Duo à 2.53Ghz

Pour ne pas compliquer la présentation de l'exemple, nous n'avons pas modélisé les lisseurs de trafic ni les files d'attente du réseau. Nous proposons une modélisation détaillée de ces éléments dans [72]. Inclure ces éléments dans le modèle augmente significativement la complexité. L'analyse d'un réseau modélisé avec des automates temporisés est limitée à des réseaux ne comportant qu'une vingtaine de flux pour le moment [4, 2, 3], mais des travaux sont en cours afin d'améliorer la méthode pour en considérer une cinquantaine.

Bien que ces techniques ne soient utilisables que sur des exemples restreints, elles ne manquent pas d'intérêt : certaines améliorations proposées sur l'*approche par trajectoire* [16] ont été permises grâce aux observations faites sur les modèles à automates temporisés, car ces modèles permettent d'exhiber le comportement pire cas du système. En revanche, pour le moment, il n'est pas envisageable d'utiliser ces techniques sur des systèmes IMA de tailles industrielles.

2.2.2 Le *tagged signal model*

Ce formalisme a été introduit par Lee et Sangiovanni-Vincentelli [75] dans le but de proposer un formalisme générique dans lequel peuvent être exprimées et comparées certaines propriétés de modèles de calcul (MoC pour *Model of Computation*) tels que les réseaux de Petri [92], les processus de Kahn [61], ...

Informellement, le *tagged signal model* représente les comportements d'un système comme une composition de *processus*. Chaque processus est défini comme l'ensemble des comportements acceptables pour un composant du système (un module, une fonction, ...). La composition de deux processus donne alors un processus défini par l'intersection des processus : cette composition est l'ensemble des comportements acceptables par les deux processus. Un comportement est défini comme un n-uplet de signaux, un *signal* étant une séquence d'*événements*. Un événement est une action significative du système comme l'activation d'une fonction, ou la production d'une occurrence d'une variable. Un événement est alors caractérisé par un couple composé d'une estampille temporelle (le *tag* de l'événement) et d'une valeur.

Vis-à-vis du problème posé, nous avons besoin à la fois de suivre l'évolution temporelle du système, mais également les dépendances entre les événements du système. Nous pouvons alors utiliser les estampilles pour suivre l'évolution temporelle et les valeurs pour suivre les relations de dépendances entre les événements. C'est la raison pour laquelle ce formalisme nous a semblé un bon candidat pour modéliser les comportements du système. Nous l'utiliserons pour fonder les bases théoriques de notre méthode.

Modélisation d une partie de l étude de cas

Pour donner l'intuition de notre formalisation dans le *tagged signal model* et pour introduire certaines notations, nous détaillons la modélisation de la fonction MFD_1 extraite de l'étude de cas. L'idée est de décrire l'ensemble des comportements possibles de la fonction à l'aide des notions d'*événements*, de *signaux* et de *processus*.

La fonction MFD_1 produit les pages à afficher (*disp*) sur l'écran du pilote et est périodiquement activée par le module M_1. A chaque activation, la fonction lit la dernière valeur reçue sur sa variable d'entrée (*wpInfo*). Après un certain traitement, elle produit une page à afficher. Le traitement dure au plus C_{MFD_1} unités de temps. Nous devons maintenant décrire comment ces événements sont liés les uns aux autres.

Evénement. Un événement est décrit à l'aide d'une estampille temporelle t et d'une valeur v définissant ses dépendances vis-à-vis des autres événements du système.

Notons $t_n^{MFD_1}$ l'estampille de l'événement de la $n^{\text{ième}}$ activation de MFD_1 et t_n^{disp} l'estampille de l'événement de la production de la $n^{\text{ième}}$ page. Comme nous faisons l'hypothèse que la fonction produit sa sortie au plus C_{MFD_1} après son activation, les estampilles de ces événements sont liées de la façon suivante :

$$t_n^{MFD_1} \leq t_n^{disp} \leq t_n^{MFD_1} + C_{MFD_1} \tag{2.1}$$

Dans notre modèle, nous capturons également les dépendances entre les événements. Par exemple, la production de la $n^{\text{ième}}$ page dépend à la fois de la $n^{\text{ième}}$ activation et du dernier événement de réception de la variable *wpInfo*. Pour transcrire ces dépendances, on associe à chaque événement une valeur en addition de son estampille. Nous définissons cette valeur comme un vecteur avec autant de composantes qu'il y a de sources d'événements. Dans notre cas, il y a trois sources d'événements : l'activation de la fonction MFD_1, la réception de *wpInfo* et la production d'un *disp*. Nous notons v_n^{disp} la valeur associée

au $n^{\text{ième}}$ événement de *disp*. Pour distinguer les différentes composantes de ce vecteur nous utilisons une notation spécifique : $v_n^{disp}{}_{MFD_1} = k$ signifie que le $n^{\text{ième}}$ événement de *disp* dépend du $k^{\text{ième}}$ événement de MFD_1 par exemple. Etant donné le comportement attendu de la fonction, nous avons pour tout $n \quad \mathbb{N}$:

$$v_n^{disp}{}_{MFD_1} = n \qquad v_n^{disp}{}_{disp} = \tag{2.2}$$

Nous utilisons la valeur pour indiquer l'absence de dépendance entre événements. Chaque instance de *disp* est produite à partir de la dernière valeur de *wpInfo* reçue. Si on suppose que la $k^{\text{ième}}$ instance de *wpInfo* est utilisée pour produire la $n^{\text{ième}}$ instance de *disp*, alors il s'agit de la dernière valeur reçue avant la $n^{\text{ième}}$ activation de MFD_1. Ce qui peut s'exprimer :

$$\begin{aligned} & v_n^{disp}{}_{wpInfo} = k \\ & \quad \text{avec } k = \max \; l \quad \mathbb{N} \quad t_l^{wpInfo} \leq t_n^{MFD_1} \end{aligned} \tag{2.3}$$

Dans notre modèle, un *événement* (t, v) est un couple composé d'une estampille temporelle et d'une valeur tel que $(t, v) \quad \mathbb{R} \times (\mathbb{N} \quad)^N$, pour un certain N. En pratique, N est le nombre de sources d'événements considérées dans le système. A noter que si une composante de la valeur d'un événement n'est pas explicitement définie, elle vaut par défaut . Nous utilisons également la notation pour une valeur ne décrivant aucune dépendance entre événements, c'est-à-dire égale à (, . . . ,).

Signal. Une séquence d'événements est appelée un *signal*. Plus précisément, un signal $\mathbf{s}$ appartient à l'ensemble $S = 2^{\mathbb{R}\times(\mathbb{N} \quad)^N}$ de tous les signaux définissables. Un N-uplet de signaux $(s_1, \ldots, s_N) \quad S^N$ définit un comportement du système. Dans cet exemple, un 3-uplet $(s_{MFD_1}, s_{wpInfo}, s_{disp}) \quad S^3$ décrit un comportement possible de MFD_1 si et seulement si tous les événements des signaux s_{MFD_1}, s_{wpInfo} et s_{disp} respectent les contraintes (2.1), (2.2) et (2.3).

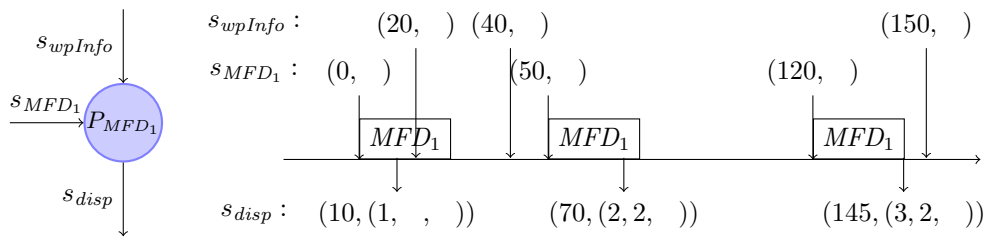
Processus. La notion de *processus* est utilisée pour regrouper l'ensemble des comportements possibles d'un élément. Par exemple, les comportements possibles de MFD_1 peuvent être résumés par le processus P_{MFD_1} tel que :

$$\begin{aligned} P_{MFD_1} = \quad & (s_{MFD_1}, s_{wpInfo}, s_{disp}) \quad S^3 \quad n \quad \mathbb{N}, \\ & t_n^{MFD_1} \leq t_n^{disp} \leq t_n^{MFD_1} + C_{MFD_1}, \\ & v_n^{disp}{}_{MFD_1} = n, \\ & v_n^{disp}{}_{wpInfo} = \max \; l \quad \mathbb{N} \quad t_l^{wpInfo} \leq t_n^{MFD_1} \quad , \\ & v_n^{disp}{}_{disp} = \end{aligned}$$

Remarque : rigoureusement, l'estampille du $n^{\text{ième}}$ événement du signal s_{MFD_1} d'un comportement $\mathbf{s} \quad S^N$ devrait par exemple s'écrire $t_n^{MFD_1}(\mathbf{s})$ pour la distinguer d'une estampille d'un autre comportement. Mais pour éviter de sur-charger les notations (qui le sont déjà bien assez), nous notons seulement $t_n^{MFD_1}$ dans la mesure où, en pratique, les estampilles de deux comportements différents ne se retrouvent jamais dans une même formule.

La figure 2.11(a) représente le système S_1 composé uniquement du processus P_{MFD_1}. Sur la figure 2.11(b) nous traçons les premiers événements des signaux d'un comportement possible de ce système. La valeur du deuxième événement du signal s_{disp} vaut $(2, 2, \quad)$ ce qui signifie que cet événement dépend du deuxième événement de s_{MFD_1} (qui correspond à la deuxième activation de MFD_1) et du deuxième événement de s_{wpInfo} (qui correspond à la deuxième réception d'une nouvelle valeur de la variable *wpInfo*).

Notons que les activations de la fonction ne sont pas nécessairement périodiques dans le processus P_{MFD_1}. L'ordonnancement de la fonction est à la charge du module où elle s'exécute. Pour imposer une période à la fonction, nous introduisons donc un deuxième processus qui modélise l'ordonnanceur du module. Ceci nous permettra d'illustrer la notion de composition de processus.



(a) Système S_1 (b) Premiers événements d'un comportement possible de S1

FIGURE 2.11 – Comportement d'un système avec P_{MFD_1}

Composition de processus. L'ordonnancement de la fonction MFD_1 est tel que : la partition de la fonction est activée périodiquement tous les T_{MFD_1} et ses activations sont décalées de O_{MFD_1} par rapport au démarrage de la MAF de M_1. En outre, les modules étant asynchrones, il existe un déphasage entre les démarrages des MAFs des modules. Nous exprimons ces déphasages vis-à-vis d'une référence commune à tous les modules : l'instant initial. Le retard par rapport à l'instant initial du démarrage de M_1 est appelé la *phase* de M_1 et est noté ϕ_{M_1}. Ainsi, l'estampille $t_n^{MFD_1}$ du $n^{\text{ième}}$ événement d'activation de MFD_1 doit respecter :

$$t_n^{MFD_1} = \phi_{M_1} + O_{MFD_1} + n \cdot T_{MFD_1} \tag{2.4}$$

Le module est également responsable de l'ordonnancement de la fonction KU_1, son processus impose une contrainte similaire sur le signal d'activation de cette fonction avec T_{KU_1} la période et O_{KU_1} l'offset de la fonction. On peut alors décrire l'ensemble des comportements possibles du module M_1 avec le processus :

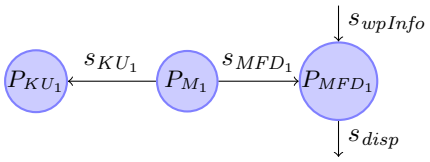
$$\begin{array}{llll} P_{M_1} = & (s_{MFD_1}, s_{wpInfo}, s_{disp}, s_{KU_1}) & S^4 & \phi_{M_1} \quad [0, HP(M_1)[\ , \quad n \quad \mathbb{N}, \\ & & & t_n^{MFD_1} = \phi_{M_1} + O_{MFD_1} + n \cdot T_{MFD_1} \\ & & & t_n^{KU_1} = \phi_{M_1} + O_{KU_1} + n \cdot T_{KU_1} \end{array}$$

avec $HP(M_1)$ l'hyper-période du module M_1, $HP(M_1) = ppcm\ T_{MFD_1}, T_{KU_1}$. On reconnait l'ordonnancement définit avec AADL et les *automates temporisés* présenté précédemment.

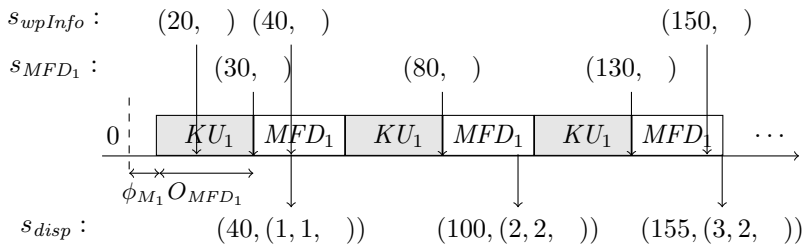
Bien que les signaux s_{wpInfo} et s_{disp} ne soient pas utilisés par le module, ils font tout de même partie de la définition du processus P_{M_1}. Ceci permet de faciliter la composition des processus : si les processus sont définis sur la même base de signaux, alors leur composition consiste en l'intersection de ces processus. Nous représentons sur la figure 2.12 le système $S_2 = P_{M_1} \setminus P_{MFD_1} \setminus P_{KU_1}$ ainsi que les premiers événements d'un comportement possible. La période T_{MFD_1} de MFD_1 vaut 50 ms et son offset O_{MFD_1} vaut 25 ms. Pour la fonction KU_1, nous avons $T_{KU_1} = 50$ ms et $O_{KU_1} = 0$. Sur l'exemple de comportement de la figure 2.11(b), la phase ϕ_{M_1} du module M_1 est égale à 5 ms. Le processus du module contraint la première activation à l'instant $\phi_{M_1} + O_{MFD_1}$, les activations suivantes s'enchaînent périodiquement tous les 50 ms. Nous n'avons pas représenté les événements d'activation de KU_1 pour ne pas alourdir la figure.
Remarque : sur les figures 2.11(a) et 2.12(a) les signaux sont représentés entrant ou sortant des processus. Ceci facilite la lecture du schéma puisque l'on y retrouve le sens des actions du système (*disp* est en sortie de MFD_1 par exemple), en revanche ces orientations n'ont pas de signification dans le *tagged signal model*.

Conclusion

Le *tagged signal model* caractérise un événement par un couple composé d'une estampille temporelle et d'une valeur. Cette valeur peut être utilisée pour encoder les relations de dépendances entre événements. Il est alors possible de « suivre » les effets d'un événement au travers du système tout en mesurant le temps écoulé entre les événements considérés. Pour cette raison, le *tagged signal model* nous a semblé être un bon candidat pour la formalisation de notre problème. En outre, la possibilité d'exprimer les



(a) Système S_2



(b) Premiers événements d'un comportement possible de S2

FIGURE 2.12 – Comportement du système composé de P_{MFD_1} et P_{M_1}

comportements du système à l'aide de contraintes ouvre la voie à l'utilisation d'outils de propagation de contraintes ou de programmation linéaire mixte pour analyser le modèle. Un modèle complet, formalisé dans le *tagged signal model*, des comportements d'un système IMA ainsi que des exigences temps réel qu'il doit satisfaire, sera défini aux chapitres 5 et 6. L'exploitation de cette formalisation, pour la vérification des exigences temps réel, sera traitée dans la troisième partie de ce manuscrit.

Dans l'exemple présenté, un temps physique est utilisé dans la mesure où cela facilite la quantification explicite des durées. Les éléments sont asynchrones mais il n'y a pas de dérives d'horloge entre eux : le temps s'écoule à la même vitesse en tout point du système et donc toutes les estampilles temporelles sont définies vis-à-vis d'une même référence temporelle basée sur le temps *physique*. A noter que certains modèles ne font pas référence au temps physique. On parle alors de temps *logique* [64]. Ces modèles permettent de vérifier des propriétés fonctionnelles de systèmes pour lesquels la notion de durée n'a pas d'importance. Le temps logique s'intéresse uniquement à caractériser des relations d'ordre entre événements : l'événement a précède l'événement b, a et b arrivent en même temps ou encore a ne peut pas être ordonné par rapport à b.

2.2.3 Clock Constraint Specification Language

Nous présentons brièvement CCSL (pour *Clock Constraint Specification Language*) qui permet de spécifier les comportements d'un modèle à l'aide de contraintes sur des horloges physiques ou logiques. Le modèle de temps de MARTE [53] est fondé sur la notion d'*horloge*. Une horloge est associée à un événement du système et un déclenchement de cette horloge représente une occurrence de l'événement. Un *événement* dans ce modèle de temps correspond donc à un *signal* du *tagged signal model* et une *occurrence* d'événement à un *événement* du *tagged signal model*. Par exemple, une horloge peut être associée à la production d'une mesure par un capteur et chaque déclenchement de l'horloge représente une nouvelle production. Ainsi chaque élément du système définit sa propre référence temporelle. Les interactions entre les éléments du système peuvent alors être représentées comme des contraintes sur leurs horloges. CCSL (pour *Clock Constraint Specification Language*) [11] a été proposé pour spécifier et analyser ces relations au sein d'un modèle MARTE. Un ensemble de relations élémentaires entre horloges permet alors d'exprimer, par composition, des comportements complexes. Une sémantique opérationnelle [10, 9] d'un sous-ensemble de CCSL a également été définie. Ainsi, il est possible de simuler l'exécution d'un modèle, notamment au sein de l'outil TIMESQUARE [56]. Une technique de traduction d'une spécification CCSL, composée d'horloges logiques, vers un modèle *Promela* a été proposée dans [110]. Ce modèle est le langage d'entrée du *model checker* SPIN [55], il est alors possible d'effectuer une vérification exhaustive de la spécification.

2.3 Evaluation de performances des réseaux embarqués

Les techniques de vérification des systèmes embarqués reposent généralement sur une vision simplifiée des réseaux : les ressources de communication ne sont considérées qu'au travers des paramètres de qualité de service qu'elles offrent, comme par exemple les meilleur et pire temps de traversée.

Ainsi, les méthodes d'évaluation de performances des réseaux embarqués ont un rôle central dans la vérification des systèmes embarqués distribués. En effet, ces méthodes permettent d'évaluer ces paramètres de qualité de service. Dans la suite, nous présentons deux méthodes d'évaluation : le *Network Calculus* et l'*approche par trajectoire*. Toutes deux ont d'ores et déjà été appliquées avec succès à l'évaluation des pires temps de traversée de réseaux AFDX [16, 18, 48, 52]. Ces méthodes permettent d'obtenir des bornes sûres sur les temps de traversée dans la mesure où, si les flux en entrée du réseau respectent leur contrat de service, la probabilité de dépasser une des bornes est nulle.

Nous citons deux autres méthodes permettant d'obtenir des bornes sûres, mais que nous n'avons pas retenues pour notre analyse globale.

- La méthode holistique [107, 102] est la première à avoir été introduite dans la littérature pour évaluer les performances des systèmes embarqués distribués. Dans le cadre de l'évaluation de performance réseau, les flux traversant le réseau sont considérés comme des tâches s'exécutant sur des processeurs qui modélisent les nœuds du réseau. Il est alors possible d'utiliser les résultats de la théorie de l'ordonnancement pour analyser le réseau. Plus précisément, l'activation d'une tâche correspond à l'arrivée d'un paquet d'un flux dans un nœud et son exécution représente la transmission du paquet. Le pire temps de réponse de la tâche est alors le retard pire cas subi par ce paquet dans le nœud. Les paramètres d'une tâche sont sa période d'activation, sa gigue et son temps d'exécution qui dépend de la taille du paquet et du débit du nœud. La période des tâches est définie par la période d'émission des paquets du flux (pour un réseau AFDX, les flux sont les VL et les périodes correspondent aux BAG). La gigue est le délai maximal qui peut retarder l'activation d'une tâche et qui est dû au retard subi par un paquet dans un nœud précédent. La gigue d'une tâche dans un nœud est définie comme étant la gigue de la tâche du flux dans le nœud précédent, à laquelle s'ajoute le pire temps de réponse dans ce nœud. Plus la gigue d'une tâche est importante, plus le nombre de paquets pouvant arriver dans un nœud pendant un intervalle de temps donné est important, ce qui augmente la charge du nœud et donc le temps de traversée de ce nœud. Pour déterminer le temps de traversée de bout en bout d'un flux, il faut alors réaliser l'analyse pire cas du temps de traversée des nœuds du réseau de proche en proche. Il a été montré dans [83] que cette méthode donne de moins bons résultats que l'*approche par trajectoire*. Nous ne l'utiliserons donc pas pour notre analyse.
- L'analyse de performances par *model checking* de réseaux embarqués a été proposée dans [44, 37] et améliorée dans [4, 3]. Comme présentées précédemment (cf. sous-section 2.2.1 page 25), ces méthodes ne sont pas utilisables pour le moment sur des configurations de tailles industrielles. Nous ne pouvons donc pas les considérer pour notre analyse.

Dans la suite, nous présentons donc plus en détails le *Network Calculus* et l'*approche par trajectoire*. En effet, Henri Bauer montre dans [18] que, dans le cadre d'un réseau AFDX, l'*approche par trajectoire* permet en moyenne de déterminer des bornes plus précises. Mais pour certains flux, le *Network Calculus* offre de meilleurs résultats. Les deux méthodes sont pessimistes dans leur évaluation, mais pas pour les mêmes raisons. Une analyse du pessimisme de l'*approche par trajectoire* a été proposée par Xiaoting Li dans [79]. Ainsi, pour obtenir des bornes les plus précises possibles, nous pouvons analyser le réseau avec les deux méthodes et sélectionner les meilleurs résultats.

Nous illustrons ces deux approches avec le modèle de réseau suivant :

Modèle de ux. Nous considérons un ensemble de flux $\tau = \ \tau_1, \ldots, \tau_n \ $. Chaque flux représente un VL. Un flux τ_i suit un chemin $\mathcal{P}_i$, qui est une séquence de nœuds, et est caractérisé par :

- b_i : le BAG du lien virtuel qui correspond au temps minimum séparant deux émissions de trames,
- s_i^m : la taille minimale d'une trame,
- s_i^M : la taille maximale d'une trame,
- $\mathcal{P}_i$: un chemin dans le réseau.

Modèle du réseau. En accord avec les hypothèses faites dans les travaux sur l'AFDX [52], chaque nœud dans le réseau sert les trames suivant une politique de service « premier arrivé premier servi » (FIFO). Le débit d'un nœud h est noté d_h, ce qui signifie que le temps d'émission d'une trame de taille s

est $\frac{s}{d_h}$. Nous considérons également que le délai de propagation entre deux nœuds est égal à une constante l, quelques soient les nœuds.

2.3.1 Network Calculus

Le *Network Calculus* a été popularisé par Le Boudec et Thiran [74]. Il permet d'évaluer des bornes déterministes sur les temps de traversée et les taux d'occupation des files d'attente d'un réseau de communication. Cette théorie est fondée sur les travaux de Cruz [40, 39] et de Chang [36] et repose sur un formalisme faisant appel à une algèbre *min-plus*. Cette algèbre correspond à la structure algébrique classique $(\mathbb{R}, +, \times)$ où l'addition est remplacée par le calcul du minimum (souvent noté) et où la multiplication est remplacée par l'addition. Ce formalisme permet de faciliter les preuves des théorèmes du *Network Calculus*.

Le *Network Calculus* utilise les notions de courbes d'arrivée pour modéliser les flux et de courbes de service pour modéliser les nœuds du réseau. Une courbe d'arrivée est une enveloppe de trafic pour un flux. L'enveloppe de trafic d'un flux en sortie d'un nœud du réseau dépend de la courbe de service du nœud et des enveloppes de trafic des flux entrant dans ce nœud. Nous illustrons cette idée sur la figure 2.13 où β_h représente la courbe de service du nœud h et pour un flux τ_i et un nœud h $\mathcal{P}_i$, nous notons α_i^h la courbe d'arrivée en entrée de h (α_i^{out} est la courbe d'arrivée en sortie du dernier nœud de $\mathcal{P}_i$). Sur cette figure, par exemple, la courbe d'arrivée α_1^3 du flux τ_1 en entrée du nœud 3 (et donc en sortie du nœud 1) dépend de la courbe de service du nœud 1, c'est-à-dire β_1, et des courbes d'arrivée α_1^1, α_2^1.

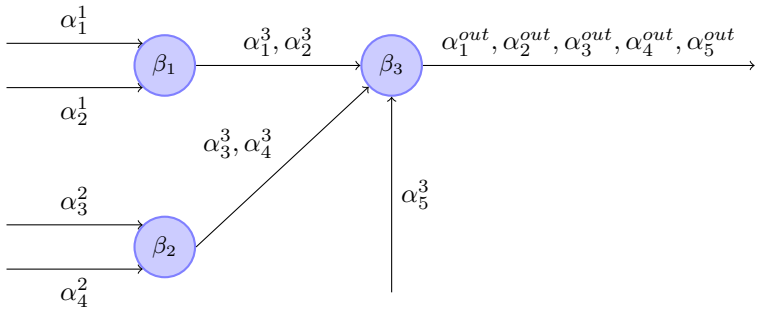


FIGURE 2.13 – Courbes d'arrivée et de service de l'exemple

Dans la suite, nous introduisons ces notions en les illustrant sur un réseau AFDX. Nous présentons *une* façon d'appliquer les résultats du *Network Calculus*. Il en existe d'autres que nous introduirons dans la conclusion de cette sous-section.

Courbes d arrivée

Dans le *Network Calculus*, un flux en entrée d'un nœud est modélisé par une courbe d'arrivée qui correspond à une enveloppe de trafic. Etant donnés un flux τ_i et un nœud h $\mathcal{P}_i$, une fonction α est une courbe d'arrivée pour τ_i si pour tout intervalle de temps $[t_1, t_2]$, le trafic apporté par τ_i pendant cet intervalle est inférieur à $\alpha(t_2 - t_1)$. Plus formellement, si on note R(t) le trafic cumulé apporté par le flux entre l'instant initial et t $\mathbb{R}^+$, alors α est une courbe d'arrivée pour le flux si et seulement si :

$$(t_1, t_2) \quad \mathbb{R}^+, \text{ t.q. } t_1 \leq t_2, \; R(t_2) - R(t_1) \leq \alpha(t_2 - t_1) \tag{2.5}$$

Nous illustrons cette définition sur la figure 2.14. La courbe de gauche représente une courbe d'arrivée α pour le trafic réel $R(t)$ de la courbe de droite. α est une courbe d'arrivée car pour tous instants t_1 et t_2, l'accroissement de R est borné par α. Autrement dit, pour tout intervalle de temps de longueur δ $\mathbb{R}^+$, le trafic maximal apporté par le flux pendant cet intervalle doit être inférieur à $\alpha(\delta)$.

La définition d'une courbe d'arrivée implique que plusieurs courbes d'arrivée sont définissables pour un même flux. Dans la suite, nous utilisons les courbes d'arrivée définies par Jérôme Grieu dans [52]. Pour un flux τ_i et un nœud h $\mathcal{P}_i$, nous notons α_i^h la courbe d'arrivée en entrée de h. Le trafic en entrée du premier nœud traversé par le flux τ_i correspond au trafic en sortie du lisseur de trafic du VL associé à τ_i.

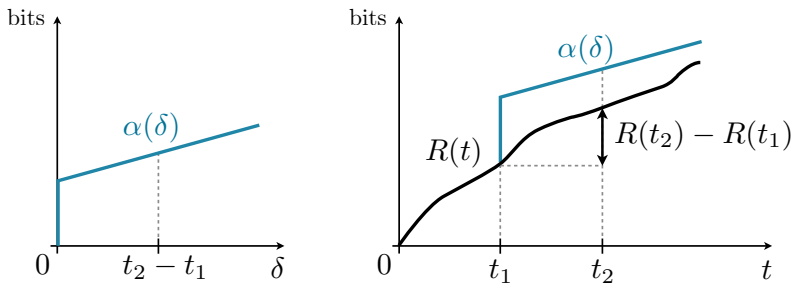


FIGURE 2.14 – Exemple d'une courbe d'arrivée

Une courbe d'arrivée pour ce trafic peut être définie par une fonction affine avec en ordonnée à l'origine s_i^M : la taille maximale d'une trame. Ceci correspond à la rafale de bits que peut émettre instantanément le VL. La pente de la fonction représente alors le débit maximal du lisseur de trafic. Le lisseur de trafic pouvant envoyer au maximum une trame de taille maximale tous les b_i, le coefficient de la pente est donc $\frac{s_i^M}{b_i}$. Cette courbe d'arrivée est définie par :

$$\delta \quad \mathbb{R}^+,\ \alpha_i^{first_i}(\delta) = \frac{s_i^M}{b_i} \cdot \delta + s_i^M \tag{2.6}$$

avec $first_i \quad \mathcal{P}_i$ le premier nœud du chemin de τ_i. La figure 2.15 représente cette courbe d'arrivée.

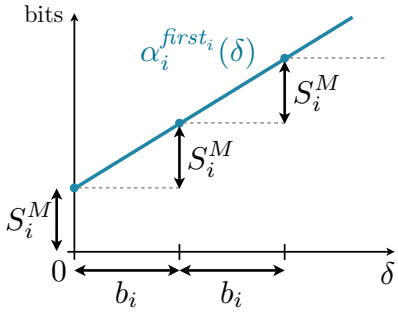


FIGURE 2.15 – Courbe d'arrivée du flux τ_i en entrée du réseau

Remarque : nous utilisons δ dans la définition de $\alpha_i^{first_i}$ pour insister sur l'idée qu'une courbe d'arrivée est définie sur des intervalles de temps et non sur des dates relatives à l'instant initial du système.

La problématique est alors de déterminer les courbes d'arrivée d'un flux tout au long de son chemin dans le réseau. Pour se faire, il faut définir comment une courbe d'arrivée est transformée en traversant un nœud du réseau, sachant que le service offert par un nœud est défini avec une courbe de service.

Courbes de service

Une courbe de service modélise le service minimum qu'offre un élément du réseau. Un nœud offre à un trafic R une courbe de service β si et seulement si β est croissante, $\beta(0) = 0$ et $R^* \geq R \otimes \beta$, avec R^* la fonction du trafic cumulé en sortie du nœud et $\otimes$ le produit de convolution tel que :

$$(R \otimes \beta)(t) = \inf_{s \leq t}(R(s) + \beta(t - s)) \tag{2.7}$$

La définition de cette notion est plus abstraite que celle des courbes d'arrivée. Nous l'illustrons avec la courbe de service retenue pour modéliser un nœud dans un réseau AFDX. Le trafic en entrée d'un nœud d'un commutateur subit systématiquement un retard de temps fixe qui correspond à la latence technologique du commutateur l (cette latence comprend le temps de propagation sur un lien et le temps

de commutation) puis le trafic est servi avec un taux constant qui est le débit du lien en sortie du nœud (100 Mbits/s dans le cas de l'AFDX). La courbe de service β_h d'un nœud h est alors définie par :

$$\beta_h(\delta) = d_h \cdot [\delta - l]^+ \tag{2.8}$$

avec $[\delta - l]^+ = \max\ 0, \delta - l$. La figure 2.16 représente cette courbe de service qui est généralement appelée *rate-latency service curve* dans la littérature.

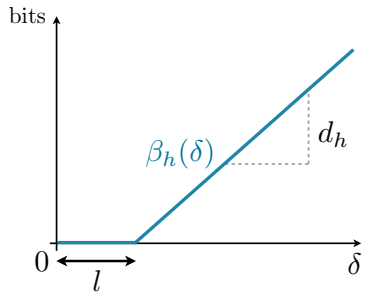


FIGURE 2.16 – Courbe de service du nœud h

Bornes sur le délai et l occupation d un n ud

Connaissant une courbe d'arrivée caractérisant l'ensemble du trafic en entrée d'un nœud et une courbe de service de ce nœud, le *Network Calculus* permet de déterminer trois bornes :

- l'occupation maximale du nœud,
- le délai maximal de traversée du nœud,
- la courbe d'arrivée en sortie du nœud.

Le trafic total en entrée d'un nœud correspond à l'agrégation des flux en entrée. La courbe d'arrivée du trafic total en entrée correspond alors à la somme des courbes d'arrivée des flux. En notant α_T^h la courbe d'arrivée pour le nœud h, on a :

$$\alpha_T^h = \sum_{i=1..card(\tau)\ h\ \ \mathcal{P}_i} \alpha_i^h \tag{2.9}$$

avec τ l'ensemble des flux du réseau.

Comme les courbes d'arrivée α_i^h sont des fonctions affines, alors α_T^h l'est également et nous notons ρ_T^h sa pente et σ_T^h son ordonnée à l'origine. Par exemple, la courbe de service considérée en entrée du nœud 1 est définie pour tout $\delta \quad \mathbb{R}^+$ par :

$$\alpha_T^1(\delta) = \alpha_1^1(\delta) + \alpha_2^1(\delta) = \frac{s_1^M}{b_1} \cdot \delta + s_1^M + \frac{s_2^M}{b_2} \cdot \delta + s_2^M = \left(\frac{s_1^M \cdot b_2 + s_2^M \cdot b_1}{b_1 \cdot b_2} \right) \cdot \delta + s_1^M + s_2^M \tag{2.10}$$

La borne maximale sur l'occupation du nœud est alors définie par la déviation maximale verticale entre la courbe de service du nœud et la courbe d'arrivée. La borne maximale sur le délai est définie par la déviation maximale horizontale. La définition de ces bornes peut se faire avec la notion de *déconvolution*, mais nous présentons une interprétation graphique de ce résultat. Sur la figure 2.17, la borne sur le délai est notée D_h et la borne sur l'occupation du nœud est notée B_h (B pour *backlog*).

L'expression arithmétique des bornes est alors :

$$\begin{aligned} D_h &= l + \frac{\sigma_h}{d_h}\ (d \text{ est tel que } \beta_h(d) = \sigma_h) \\ B_h &= \alpha_T^h(l) = \rho_h \cdot l + \sigma_h \end{aligned} \tag{2.11}$$

On détermine ainsi le pire retard que peut subir l'agrégation de tous les flux en entrée de h, et il est donc possible de garantir qu'aucun des flux, individuellement, ne subira un retard supérieur à D_h.

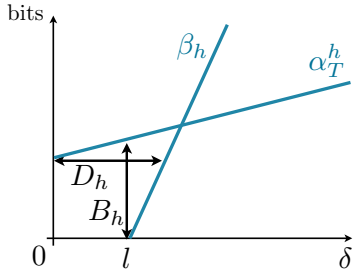


FIGURE 2.17 – Interprétation graphique des bornes de l'occupation d'un nœud et du délai

Courbe d arrivée en sortie d un n ud

Pour déterminer la courbe d'arrivée de chacun des flux en sortie du nœud, l'idée est alors de considérer que chacun des flux a subi un retard égal à D_h. Graphiquement, ceci revient à décaler la courbe d'arrivée en entrée de D_h vers la gauche, comme illustré sur la figure 2.18.

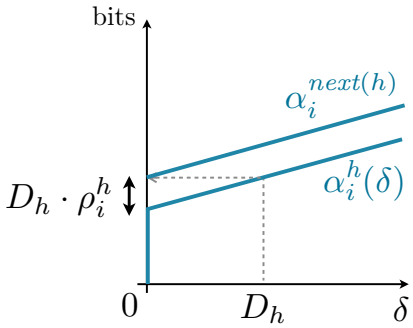


FIGURE 2.18 – Courbe d'arrivée d'un flux en sortie d'un nœud

La courbe d'arrivée $\alpha_i^{next_i(h)}$ d'un flux τ_i en sortie du nœud h, et donc en entrée du nœud suivant $next_i(h)$, est alors une fonction affine et pour tout $\delta \quad \mathbb{R}^+$:

$$\alpha_i^{next(h)}(\delta) = \rho_i^h \cdot \delta + (D_h \cdot \rho_i^h + \sigma_i^h) \tag{2.12}$$

avec ρ_i^h la pente de la courbe d'arrivée du flux τ_i en entrée de h et σ_i^h son ordonnée à l'origine.

La traversée d'un nœud a donc pour effet d'augmenter la rafale de la courbe d'arrivée. En revanche, la pente de la courbe n'est pas modifiée. En disposant de cette courbe d'arrivée, il est alors possible de propager le calcul jusqu'au nœud suivant et ainsi déduire l'ensemble des délais pire cas dans tous les nœuds.

Délai pire cas de traversée du réseau

Le pire délai de traversée du réseau pour un flux τ_i est alors la somme des délais pire cas de chacun des nœuds du chemin $\mathcal{P}_i$. En notant $WCTT_i$ (pour *Worst Case Traversal Time*) le pire délai de traversée du réseau pour le flux τ_i, on a :

$$WCTT_i = \sum_{h \quad \mathcal{P}_i} D_h \tag{2.13}$$

Nous rappelons qu'il s'agit ici d'un majorant du pire délai de traversée. Ce délai pouvant ne jamais être atteint par le système réel.

Conclusion

Le *Network Calculus* propose un cadre rigoureux pour l'analyse des délais pire cas dans les réseaux. En outre, certains résultats sont facilement applicables aux réseaux AFDX. A noter que nous avons présenté *une* façon d'appliquer les résultats de cette théorie, et qu'il en existe d'autres. En particulier, Jérôme Grieu propose dans sa thèse la notion de *groupe* de flux qui permet de prendre en compte la sérialisation des flux en sortie d'un nœud. Plus précisément, les paquets des flux sortent les uns à la suite des autres du nœud et, si le nœud suivant a le même débit, alors les paquets ne se retardent plus. Considérer cet effet ne complique pas significativement les algorithmes mais offre de meilleurs résultats.

La méthode dite *Pay Multiplexing Only Once* [97] utilise d'autres résultats du *Network Calculus* pour offrir de meilleurs résultats, mais au prix d'une plus grande complexité.

Dans [27], les auteurs utilisent le *Network Calculus* pour définir les contraintes de programmes linéaires dont la résolution permet de déterminer une borne précise du temps de traversée. Cependant cette approche souffre de deux limitations par rapport à nos objectifs : (1) la politique de service considérée est *blind multiplexing*, c'est-à-dire qu'aucune hypothèse n'est faite sur l'ordre de traitement des paquets, mis à part que les paquets d'un même flux sont traités dans leur ordre d'arrivée, et (2) la complexité de la méthode fait qu'elle n'est applicable que sur des études de cas de petite dimension, ou sur une topologie réseau particulière dite *tandem*.

Dans [30], les auteurs proposent une méthode de collaboration entre théorie de l'ordonnancement et le *Network Calculus*. La méthode permet de déterminer les courbes d'arrivée de flux en entrée du réseau, tout en prenant en compte l'ordonnancement des tâches émettrices. Les courbes d'arrivée modélisent ainsi plus finement le trafic réel, ce qui permet de limiter le pessimisme.

Un des intérêts du *Network Calculus* réside dans l'outillage disponible :

- *COINC* [26] est une librairie pour Scilab permettant la manipulation de courbes d'arrivée et de service avec les opérations classiques du *Network Calculus*,
- *DISCO Network Calculator* [97] est une librairie *Java* permettant d'analyser un réseau suivant différentes approches fondées sur le *Network Calculus*. En outre, cette librairie contient des algorithmes permettant de vérifier que la topologie du réseau respecte certaines propriétés,
- *NC-Maude* [29] qui, contrairement aux autres outils, ne fonctionne pas comme une « boite noire », c'est-à-dire qu'il ne suffit pas de lui fournir en entrée une configuration de réseau pour obtenir les pires temps de traversée de ses flux. Cet outil propose une approche pédagogique dans le sens où il oblige la manipulation directe des formules du *Network Calculus* pour obtenir un résultat.

2.3.2 Approche par trajectoire

L'*approche par trajectoire* a été proposée par Steven Martin [83]. Tout comme l'*approche holistique*, l'*approche par trajectoire* se fonde sur les principes de la théorie de l'ordonnancement, mais contrairement au *Network Calculus* ou à l'approche holistique, cette méthode considère le pire cas d'un flux sur l'ensemble de son chemin, et non pas nœud par nœud.

Un flux τ_i en entrée du réseau est caractérisé par la distance minimale entre deux activations de tâches, ce qui correspond à la distance minimale entre deux arrivées de paquets et donc au BAG b_i. Une propriété importante d'un flux est le temps maximal de transmission pour chacun des noeuds qu'il traverse. Pour chaque nœud h de débit d_h, le temps de transmission d'un paquet du flux τ_i est borné par : $C_i^h = \frac{s_i^M}{d_h}$, où s_i^M est la taille maximale d'un paquet de τ_i.

Principes

Considérons le scénario de la figure 2.19. Les périodes actives des nœuds 1,2 et 3 sont représentées. Les périodes de transmission des paquets sont notées m_{ij} avec i l'indice du flux et j l'indice d'un paquet. Par exemple, m_{42} est le 2ième paquet du flux τ_4. L'instant d'arrivée d'un paquet m_{ij} dans un nœud h est noté $a_{m_{ij}}^h$.

Sur la figure, on détermine le temps de traversée R_{12} du paquet m_{12}, qui est défini par :

$$R_{12} = W_{12}^3(t) - t + C_1^3 \tag{2.14}$$

avec t l'instant d'arrivée de m dans le réseau, $W_{12}^3(t)$ l'instant où le nœud 3 commence à transmettre m_{12} et C_1^3 le temps nécessaire à cette transmission. L'instant initial est alors défini comme l'instant d'arrivée du premier paquet interférant avec m_{12}. Il s'agit dans cet exemple du paquet m_{21}. A noter que ce paquet

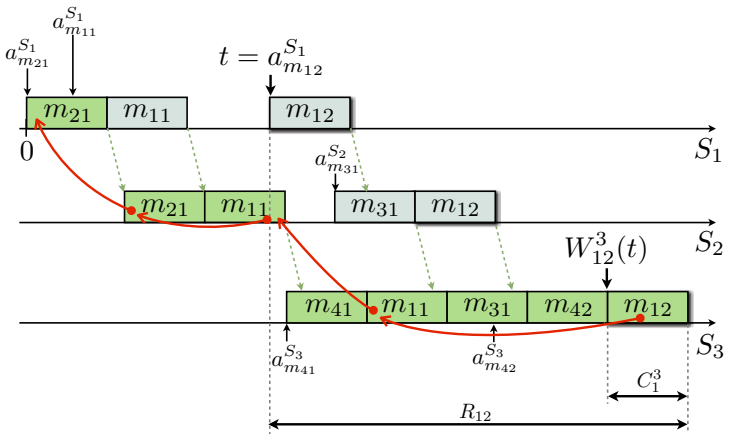


FIGURE 2.19 – Principes de l'approche par trajectoire

n'interfère pas directement avec m_{12} mais il retarde m_{11} dans le nœud 1 qui ensuite retarde m_{12} dans le nœud 3. L'ensemble des paquets impliqués dans le calcul de R_{12} sont mis en valeur. On peut remarquer par exemple que le paquet m_{11} intervient deux fois dans la valeur de $W^3_{12}(t)$. $W^3_{12}(t)$ est interprétée comme une chaîne de périodes actives, remontant du dernier nœud traversé jusqu'au premier paquet interférant avec m.

Pour déterminer le pire temps de traversée du réseau d'un flux τ_i, l'idée est d'identifier l'ensemble des paquets pouvant interférer avec un paquet m, choisi arbitrairement, de ce flux. La démarche est alors de déterminer une date de démarrage au plus tard sur le dernier noeud traversé, valable pour tout paquet du flux i arrivant à une date t. Cet instant est noté $W_i^{last_i}(t)$ et le pire temps de traversée du réseau pour ce flux est alors défini par :

$$R_i = \max_{t \quad \mathbb{R}^+} \; W_i^{last_i}(t) - t + C_i^{last_i} \tag{2.15}$$

Pour chaque flux τ_j croisant le flux τ_i, il est possible de définir une fenêtre d'interférence, c'est-à-dire un intervalle de temps pendant lequel tous les paquets produits par τ_j sont susceptibles d'interférer avec le paquet m. Notons $first_{ij}$ le premier nœud où les flux τ_i et τ_j se croisent. Comme la politique de service est « premier arrivé premier servi »alors seuls les paquets de τ_j arrivant avant m dans $first_{ij}$ peuvent retarder m sur son chemin. En effet, un paquet arrivant après m dans $first_{ij}$ ne pourra pas retarder m dans ce nœud et arrivera nécessairement après m dans le nœud suivant. Il ne pourra donc pas retarder m dans ce nœud non plus, et ainsi de suite. En notant $S^h_{max_i}$ le pire temps requis pour un paquet du flux τ_i pour rejoindre le nœud h, alors seuls les paquets de τ_j arrivant avant $t + S^{first_{ij}}_{max_i}$ peuvent interférer avec m.

Dans [83], il a été montré que pour chaque nœud sur le chemin de m, il existe une borne avant laquelle aucun paquet ne peut retarder m. Pour un nœud $h \quad \mathcal{P}_i$, cette borne est notée M_i^h et est telle que :

$$M_i^h = \sum_{h' \quad prec(h,\mathcal{P}_i)} \min_{j,h' \quad \mathcal{P}_j} C_j^{h'} \; + card(prec(h,\mathcal{P}_i)) \cdot l \tag{2.16}$$

avec $prec(h, \mathcal{P}_i)$ l'ensemble des nœuds précédents h sur le chemin $\mathcal{P}_i$ et $card(s)$ le cardinal de l'ensemble s. Ainsi, tous les paquets de τ_j interférant avec m arrivent dans le nœud $first_{ij}$ pendant l'intervalle $[M_i^{first_{ij}}, t + S^{first_{ij}}_{max_i}]$. Comme illustré sur la figure 2.20, pour arriver dans cet intervalle, un paquet τ_j doit être produit après $M_i^{first_{ij}} - S^{first_{ij}}_{max_j}$ et avant $t + S^{first_{ij}}_{max_i} - S^{first_{ij}}_{min_j}$, avec $S^{first_{ij}}_{min_j}$ le temps minimum requis par un paquet de τ_j pour rejoindre le nœud $first_{ij}$.

La fenêtre d'interférence du flux τ_j sur le flux τ_i est alors l'intervalle :

$$\left[M_i^{first_{ij}} - S^{first_{ij}}_{max_j}, t + S^{first_{ij}}_{max_i} - S^{first_{ij}}_{min_j}\right] \tag{2.17}$$

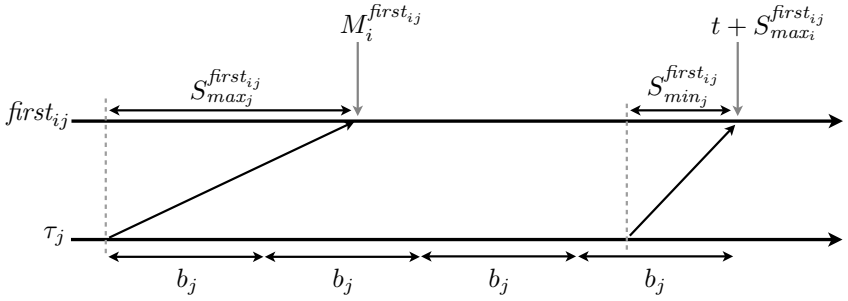


FIGURE 2.20 – Fenêtre d'interférence de τ_j sur τ_i

Le flux τ_j produit au maximum un paquet tous les b_j, nous pouvons donc déduire le nombre maximal de paquets de τ_j intervenant dans le calcul de $W_i^{last_i}(t)$:

$$\begin{aligned} &1 + \left\lfloor \frac{t + A_{ij}}{b_j} \right\rfloor \\ &\text{avec } A_{ij} = S_{max_i}^{first_{ij}} - S_{min_j}^{first_{ij}} - M_i^{first_{ij}} + S_{max_j}^{first_{ij}} \end{aligned} \tag{2.18}$$

Dans le pire des cas, tous ces paquets interviennent dans $W_i^{last_i}(t)$ et sont comptabilisés dans le nœud le plus lent sur $\mathcal{P}_i \setminus \mathcal{P}_j$. Ce nœud est noté $slow_{ij}$ et la contribution de ces paquets est alors :

$$\left(1 + \left\lfloor \frac{t + A_{ij}}{b_j} \right\rfloor\right) \cdot C_j^{slow_{ij}} \tag{2.19}$$

Pour maximiser le terme $W_i^{last_i}(t)$, les périodes actives le composant sont alignées comme représentées sur la figure 2.21, c'est-à-dire que le démarrage d'une période correspond à l'arrivée du dernier paquet du nœud précédent. Ainsi, certains paquets doivent apparaître plusieurs fois dans l'estimation de $W_i^{last_i}(t)$. Sur la figure, les nœuds du chemin $\mathcal{P}_i$ sont numérotés de 1 à q et $slow_i$ est le nœud le plus lent sur ce chemin. Pour tout nœud h, $f(h)$ est le premier paquet (f pour *first*) a être comptabilisé dans la période active du nœud h. Les paquets comptabilisés deux fois dans $W_i^{last_i}(t)$ sont mis en valeur.

Dans [83], cet effet est majoré en considérant que les paquets comptés deux fois sont systématiquement les paquets dont le temps de transmission est le plus long, ce qui est exprimé pour l'ensemble des nœuds de $\mathcal{P}_i$ par :

$$\sum_{h \ \mathcal{P}_i \ slow_i} \max_{j,h \ \mathcal{P}_j} C_j^h \tag{2.20}$$

Ce qui mène à une expression du terme $W_i^{last_i}(t)$ telle que :

$$W_i^{last_i}(t) = \sum_{j,\mathcal{P}_j \setminus \mathcal{P}_i =} \left(1 + \left\lfloor \frac{t + A_{ij}}{b_j} \right\rfloor\right) \cdot C_j^{slow_{ij}} + \sum_{h \ \mathcal{P}_i \ slow_i} \max_{j,h \ \mathcal{P}_j} C_j^h + (card(\mathcal{P}_i) - 1) \cdot l - C_i^{last_i} \tag{2.21}$$

avec $A_{ij} = S_{max_i}^{first_{ij}} - S_{min_j}^{first_{ij}} - M_i^{first_{ij}} + S_{max_j}^{first_{ij}}$. Le terme $(card(\mathcal{P}_i) - 1) \cdot l$ permet de prendre en compte la latence technologique entre chaque nœud du chemin $\mathcal{P}_i$. Le temps de transmission $C_i^{last_i}$ est soustrait car $W_i^{last_i}(t)$ représente le démarrage de la transmission du paquet m, et non pas la fin de sa transmission.

Délai pire cas de traversée du réseau

Les formules (2.15) et (2.21) donnent alors le pire temps de traversée du réseau pour le flux τ_i, qui est égal à :

$$WCTT_i = \max_{t \ \mathbb{R}^+} W_i^{last_i}(t) - t + C_i^{last_i} \tag{2.22}$$

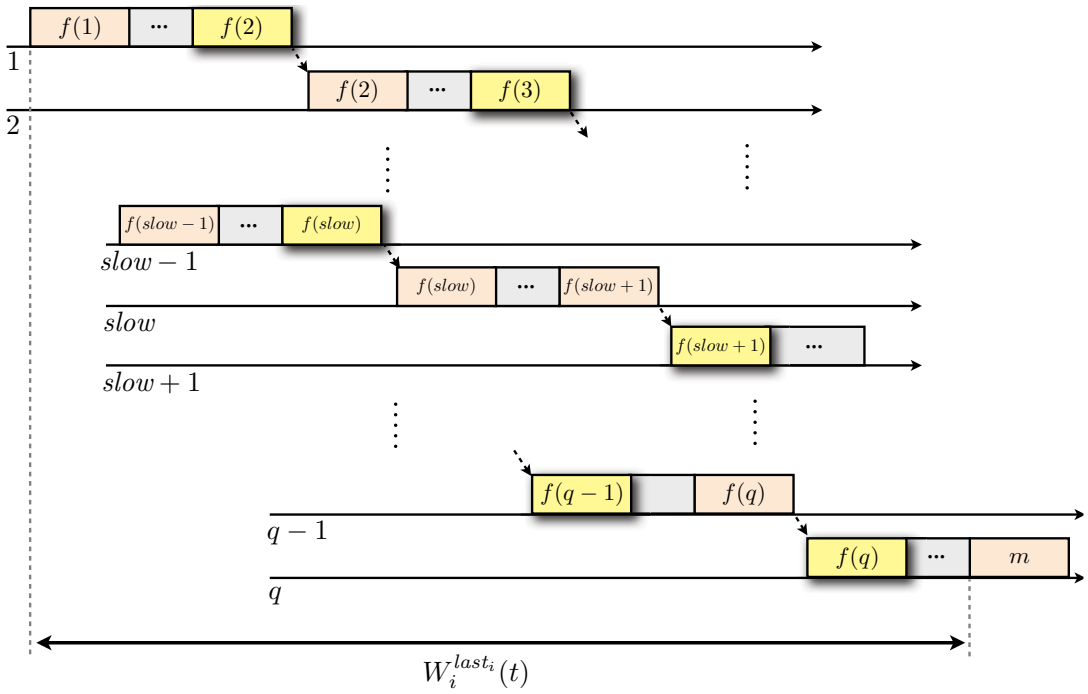


FIGURE 2.21 – Paquets devant être comptabilisés deux fois dans $W_i^{last_i}(t)$

En pratique, ce maximum est facile à déterminer dans la mesure où $W_i^{last_i}(t)$ évolue par palier en fonction de t.

Conclusion

L'*approche par trajectoire* définit une méthode efficace pour le calcul des bornes de temps de traversée sûres sur les réseaux AFDX. Cette méthode a été améliorée dans [18] pour prendre en compte la notion de groupage proposée par Jérôme Grieu dans sa thèse.

Il n'existe pour le moment pas d'outils *open source* implémentant ces algorithmes. Cependant, dans le cadre du projet SATRIMMAP, nous avons développé un outil d'analyse réseau fondé sur cette approche, ce qui nous a permis de tester sa robustesse vis-à-vis du passage à l'échelle. L'analyse complète du réseau AFDX de l'A380 requiert entre 4 et 5 secondes de calcul, ce qui est tout à fait raisonnable dans la mesure où ce réseau est constitué d'environ 1000 VL.

Dans l'*approche par trajectoire* ou le *Network Calculus* tous les flux sont considérés asynchrones, c'est-à-dire qu'il n'existe pas de corrélation entre les dates d'arrivée dans le réseau des paquets de deux flux distincts. Or, dans un module IMA, les différents émetteurs (les partitions) s'exécutent séquentiellement dans des tranches de temps prédéfinies. Ainsi, les émissions des flux en sortie d'un module sont corrélées car pendant qu'un émetteur est actif, les autres ne le sont pas. Dans le cadre des systèmes IMA, le trafic réel en sortie d'un module est donc moins important que celui considéré dans l'*approche par trajectoire* ou le *Network Calculus*. C'est la raison pour laquelle, dans cette thèse, nous nous sommes orientés vers l'*approche par trajectoire* car il nous a semblé plus facile de modifier cette méthode pour tirer parti de cette observation. Nous avons présenté ces résultats dans [72]. Les auteurs de [78] ont également fait cette observation et proposent une méthode fondée sur le *Network Calculus*.

2.4 Evaluation d'exigences de bout en bout

Après avoir passé en revue les techniques spécifiques d'évaluation de performances des réseaux embarqués, nous nous intéressons aux techniques plus générales d'évaluation d'exigences de bout en bout. Dans la littérature, nous trouvons principalement des travaux traitant de la latence, notamment des approches fondées sur la théorie de l'ordonnancement ou sur le *Real-Time Calculus*. Les notions de fraîcheur ou de

cohérence sont plus rarement étudiées dans le domaine des systèmes embarqués. Cet état de l'art fait par conséquent référence à un autre domaine s'intéressant à ces notions : les bases de données temps réel.

2.4.1 Evaluation de latence

Outre les analyses réalisables par *model checking*, on trouve principalement deux types d'approches pour l'évaluation de latence de bout en bout dans les systèmes distribués. Le premier type concerne les approches fondées sur la théorie de l'ordonnancement, tandis que le deuxième est fondé sur l'utilisation du *Real-Time Calculus*, qui est une généralisation du *Network Calculus*.

Approche théorie de l ordonnancement

De nombreuses méthodes d'analyse de bout en bout de systèmes distribués ont été proposées sur la base des travaux fondateurs de Liu et Layland [80]. Les systèmes considérés dans ces travaux sont généralement constitués d'un ensemble de nœuds de calcul (des processeurs) sur lesquels s'exécutent périodiquement des tâches. Une chaîne de traitement est alors représentée comme une séquence de tâches, le démarrage de chaque tâche étant conditionné par la terminaison de la tâche précédente. Ces modèles ne permettent pas de prendre en compte l'ensemble des comportements des systèmes IMA que nous voulons inclure dans notre analyse. Ils sont cependant pertinents dans un contexte plus général et sont le fondement théorique de l'*approche par trajectoire*.

La première méthode s'intéressant spécifiquement aux systèmes distribués est la méthode holistique [102, 107], que nous avons présentée dans la cadre de l'évaluation de performance réseau. Le pire temps de réponse d'une séquence de traitement est déterminé en additionnant les pires temps de réponse de chacune de ces tâches. Le pire temps de réponse d'une tâche est calculé en fonction du pire temps de réponse de la tâche précédente. Le pire temps de la chaîne de traitement a alors tendance à « exploser » en fonction du nombre de tâches. Pour minimiser le pessimisme de cette approche, les modèles à *o set* ont été introduits [88, 81]. Les *o sets* permettent de modéliser plus finement les relations de précédence qui existent entre les tâches d'une même chaîne.

Les auteurs de [58, 59] proposent une approche originale pour l'analyse des systèmes distribués. L'idée est de transformer cette analyse en un problème équivalent dans un contexte non distribué (ou monoprocesseur), dans lequel des algorithmes classiques [14] peuvent être utilisés pour déterminer les pires temps de réponse. Si ces temps de réponse sont corrects vis-à-vis des exigences alors ils le sont également sur le problème initial. Les arguments utilisés lors de la construction des paramètres du problème monoprocesseur équivalent sont similaires à ceux utilisés dans l'*approche par trajectoire*. Les auteurs montrent que cette méthode offre de meilleurs résultats que l'approche holistique pour des chaînes de traitement de plus de 5 nœuds. Il est possible de prendre en compte différentes politiques de service, mais l'ensemble des nœuds doit partager la même politique.

Real-Time Calculus

Le *Real-Time Calculus* a été proposé par Thiele, Chakraborty et Naedele [104] comme une généralisation du *Network Calculus* (présenté à la sous-section 2.3.1 page 35) permettant l'analyse de performances des systèmes temps réel. Bien qu'il existe des travaux sur des extensions du *Real-Time Calculus* vers les domaines stochastiques [96], il s'agit fondamentalement d'une théorie déterministe dans le sens où la probabilité que les performances calculées soient dépassées est nulle.

Principes. Le *Real-Time Calculus* permet de modéliser l'utilisation des ressources des composants d'un système. Le comportement d'un composant est caractérisé par la notion de *courbe d'arrivée* qui représente la charge de travail qui lui est envoyée et la notion de *courbe de service* qui représente les ressources dont il dispose. Un tel composant est représenté sur la figure 2.22. Nous prenons en exemple la modélisation de la fonction de l'étude de cas FM_1, qui génère des requêtes à la base de données pour fournir les informations du *waypoint* demandées par le pilote. Cette modélisation est inspirée par les travaux effectués sur les ordonnancements de type TDMA (pour *Time Division Multiple Access*) [108].

Une particularité du *Real-Time Calculus* par rapport au *Network Calculus* est la modélisation des comportements pire cas *et* meilleur cas. En entrée du composant, α^u_{wpId} et α^l_{wpId} sont les courbes d'arrivée respectivement pire (u pour *upper*) et meilleur cas (l pour *lower*). Elles représentent alors le nombre d'occurrences de la variable *wpId* pouvant être délivrées pendant tout intervalle de temps. Ces courbes sont représentées sur la figure 2.23. Par exemple, pour tout intervalle de temps de longueur 80, le nombre

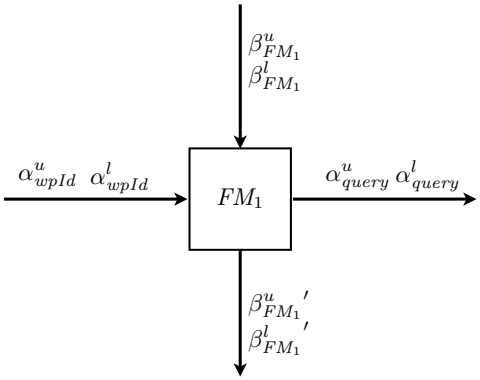


FIGURE 2.22 – Un composant du *Real-Time Calculus*

d'occurrences de la variable *wpId* arrivant dans FM_1 est compris entre 1 et 4. Nous avons tracé ces courbes de façon arbitraire, mais elles pourraient être déduites en fonction des composants précédant FM_1.

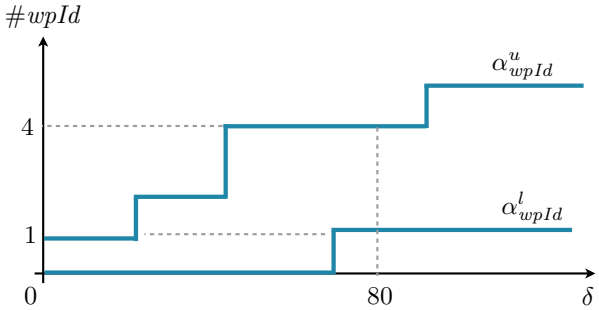


FIGURE 2.23 – Courbes d'arrivée de *wpId* en entrée de FM_1

Les autres caractéristiques en entrée de FM_1 sont β^u et β^l. Il s'agit des courbes de service respectivement pire et meilleur cas. Elles représentent le nombre d'occurrences de *wpId* que la fonction peut traiter pendant tout intervalle de temps. Ces courbes sont représentées sur la figure 2.24. Nous traçons ces courbes de la façon suivante : la fonction acquiert ses entrées en début de période, et peut traiter n occurrences à chaque période. En outre, la fonction produit toutes les occurrences de *query* au même instant, cet instant pouvant intervenir n'importe quand à l'intérieur de la partition. Ainsi, dans le pire cas, la fonction ne fournit aucun service pendant un intervalle de temps de longueur $T_{FM_1} + C_{FM_1}$ et ensuite traite 2 occurrences toutes les périodes (courbe $\beta^l_{FM_1}$). Dans le meilleur cas, la fonction traite instantanément 2 occurrences et pourra en traiter 2 à nouveau à la période suivante ($\beta^u_{FM_1}$).

Le *Real-Time Calculus* permet alors de déterminer la latence pire cas subie par une occurrence de *wpId*. Graphiquement, cette latence est obtenue en déterminant la distance maximale séparant la courbe d'arrivée pire cas α^u_{wpId} avec la courbe de service pire cas $\beta^u_{FM_1}$. Cette latence pire cas est notée l_{FM_1} sur la figure 2.25. A noter que le maximum de déviation verticale entre ces courbes permet d'obtenir le nombre maximal d'occurrences en attente de traitement. On peut donc déduire de la figure que le nombre maximal d'occurrences de *wpId* en attente est 2.

Un autre résultat important du *Real-Time Calculus* est de déterminer les courbes d'arrivée et de service en sortie d'un composant. Il s'agit de α^u_{query}, α^l_{query}, $\beta^u_{FM_1}$, $\beta^l_{FM_1}$ sur la figure 2.22. Il est ainsi possible d'analyser les performances d'un réseau de composant tel que représenté sur la figure 2.26. Nous n'avons pas inclus dans cet exemple les composants nécessaires à la modélisation des éléments du réseau. A noter que les courbes de service en sortie des composants ne sont pas représentées car elles n'ont pas d'utilité dans cet exemple.

L'analyse est alors réalisée de façon itérative. Connaissant les courbes d'arrivée de la variable *req* et les courbes de service du KU_1, il est possible de déterminer la latence l_{KU_1} ainsi que les courbes d'arrivée de *wpId*. Avec ces courbes d'arrivée, il est alors possible de déterminer la latence l_{FM_1} et les courbes

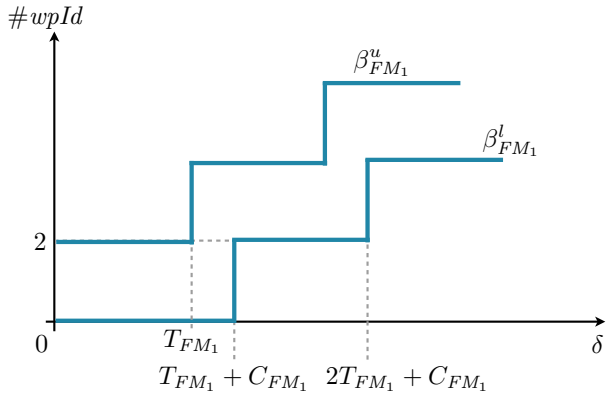


FIGURE 2.24 – Courbes de service de FM_1

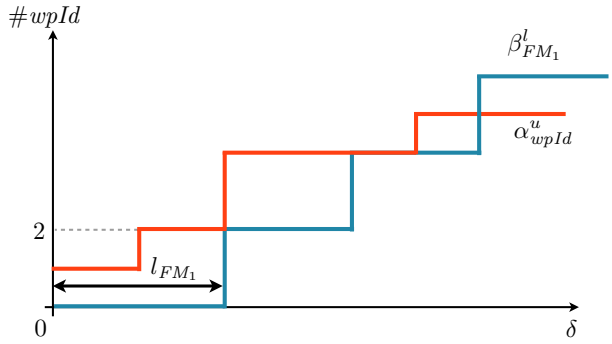


FIGURE 2.25 – Pire latence d'une occurrence de *wpId*

d'arrivée de *wpInfo* ce qui finalement permet de déterminer la latence l_{MFD_1}. La latence de bout en bout est alors la somme des latences de chaque composant, soit $l_{tot} = l_{KU_1} + l_{FM_1} + l_{MFD_1}$.

Lors d'une analyse de bout en bout, la modélisation des comportements à l'aide d'enveloppes mène à des résultats pessimistes. En effet, les courbes d'arrivée ou de service doivent être valables pour n'importe quel intervalle de temps débutant à n'importe quel instant. Ainsi, il n'est pas possible de prendre en compte des corrélations entre événements. En particulier, les démarrages des occurrences de MFD_1 dépendent des démarrages de KU_1 car les deux fonctions sont ordonnancées statiquement sur un même module. Ainsi, si KU_1 est dans une configuration pire cas pour le traitement d'une occurrence de *req*, alors MFD_1 ne peut pas être dans une configuration pire cas pour traiter l'occurrence de *wpInfo* qui dépend de cette requête. Or, le *Real-Time Calculus* ne peut pas prendre en compte cet effet, ce qui engendre un pessimisme dans l'analyse. Ceci est illustré sur la figure 2.27. La figure 2.27(a) décrit un scénario pire cas pour le traitement d'une requête *req*[*i*] tel que considéré par le *Real-Time Calculus* : dans le pire cas, les fonctions ne fournissent aucun service pendant une période et un temps d'exécution maximal. Sur la figure 2.27(b), un scénario prenant en compte les corrélations entre les démarrages de KU_1 et MFD_1 est représenté. On constate alors que si KU_1 et MFD_1 sont dans une configuration pire cas, alors MFD_1 ne peut pas réaliser son pire cas.

Conclusion. Le *Real-Time Calculus* a été proposé pour la modélisation et l'analyse de systèmes distribués, notamment du point de vue de l'utilisation de ressources. Cette théorie fournit des résultats permettant de déterminer les temps de traversée des composants d'un système, ainsi que leur occupation maximale. L'Ecole Polytechnique Fédérale de Zurich (ETHZ) met à disposition une suite d'outils Matlab [109] implémentant les algorithmes du *Real-Time Calculus*. Du fait de la modélisation par enveloppe, cette technique est pessimiste, mais elle permet d'obtenir des résultats bien plus rapidement que des analyses par *model checking* [63].

Nous n'utiliserons pas cette approche pour la résolution de notre problème pour principalement deux

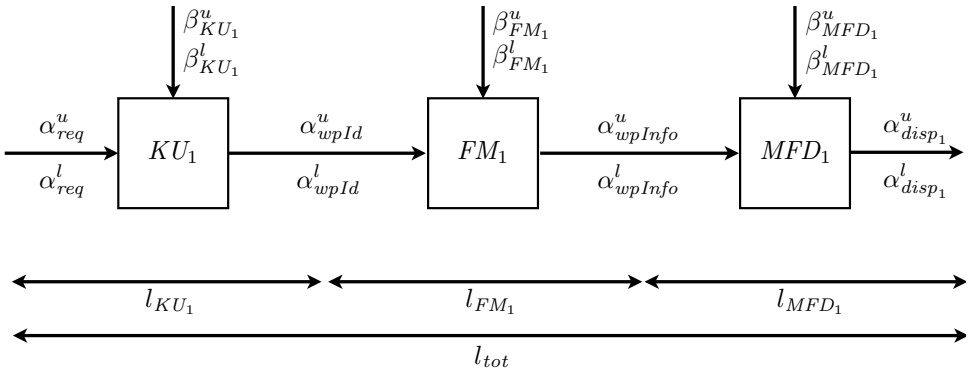


FIGURE 2.26 – Latence de bout en bout

raisons. Comme illustré par la figure 2.27, il n'est pas possible de prendre en compte les spécificités de l'ordonnancement des modules IMA dans une analyse de bout en bout. Ceci induit des sur-approximations et notre objectif est de limiter autant que possible le pessimisme de l'analyse. L'autre raison, plus fondamentale, est que les notions de fraîcheur et de cohérence ne sont pas définies ou exprimables dans le *Real-Time Calculus* car les dépendances ou corrélations entre événements ne sont pas modélisables. Cette approche ne pourrait par conséquent que répondre partiellement à notre problème.

A noter que les auteurs de [100] suggèrent que les algorithmes du *Real-Time Calculus* souffrent d'un problème de passage à l'échelle quand il existe des différences importantes entre les échelles de temps utilisées par les composants. Notre étude de cas est concernée par cette limitation. En effet, les périodes des fonctions sont exprimées en dizaines de milli-secondes mais la transmission d'une trame réseau s'exprime en dizaines de micro-secondes.

2.4.2 Evaluation de fraîcheur

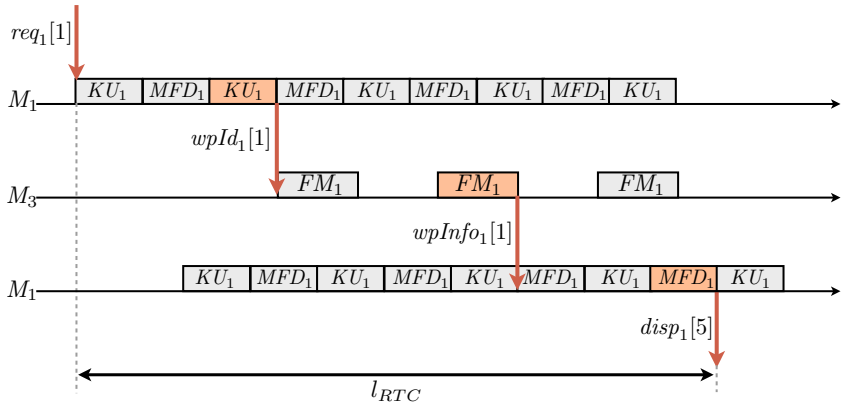
La fraîcheur est une notion classique dans le domaine des bases de données temps réel [91]. De nombreux travaux spécifiques à ce domaine sont disponibles. Cependant, dans ces travaux la fraîcheur d'une donnée peut avoir différentes significations. Dans [28], les auteurs proposent une taxonomie des notions de fraîcheur utilisées. La fraîcheur d'une donnée peut être définie suivant deux facteurs :

- *Currency* s'intéresse au temps écoulé entre la modification d'un objet du monde réel et la mise à jour de la donnée le représentant. Cette notion peut se rapprocher de la notion de latence que nous utilisons.
- *Timeliness* s'intéresse à l'âge d'une donnée dans la base, c'est-à-dire au temps qui peut s'écouler entre deux mises à jour de la donnée. Cette notion correspond à la notion de fraîcheur que nous utilisons.

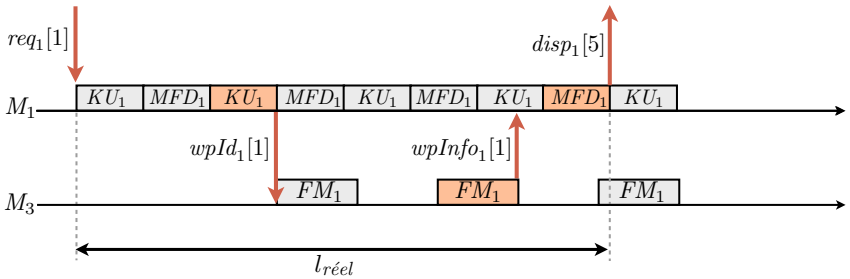
En général, ces travaux ont pour objectif de déterminer un ordonnancement des tâches du système de gestion de base de données qui minimise l'utilisation des ressources, tout en garantissant que la fraîcheur des données reste acceptable. L'efficacité de l'ordonnancement est alors évaluée par simulation.

Dans le domaine des systèmes embarqués, on retrouve la notion de fraîcheur dans [82]. Les auteurs traitent un problème de génération de configuration pour une plateforme automobile. Des tâches applicatives sont réparties sur un ensemble de calculateurs, appelés ECU (pour *Electronic Control Unit*). Les tâches applicatives s'échangent des données. Les communications entre ECU sont gérées par un *middleware* dont le rôle est de regrouper les données en trames réseau avant de les transmettre. Sur chaque ECU, les tâches du *middleware* sont en concurrence avec les tâches applicatives. La méthode proposée génère une configuration qui définit les regroupements de donnée en trame ainsi que les caractéristiques des tâches du *middleware*. La configuration déterminée doit garantir qu'un ensemble d'exigences de fraîcheur sur les données sont satisfaites.

La pire fraîcheur d'une donnée est définie comme l'âge maximal que peut atteindre une occurrence de la donnée. Une occurrence de donnée « vieillit » entre le moment où la tâche la produisant est activée et pendant l'intervalle de temps où elle peut être utilisée par la tâche l'ayant consommée, c'est-à-dire jusqu'au moment où l'occurrence suivante de la donnée est consommée par cette tâche. Une autre contrainte du problème est de garantir que les données produites ne sont jamais écrasées, ce qui garantit que l'occurrence



(a) Scénario pire cas considéré avec le *Real-time Calculus*



(b) Scénario pire cas réel

FIGURE 2.27 – Exemple de pessimisme de l'analyse de bout en bout avec le *Real-time Calculus*

suivante arrivera bien jusqu'à la tâche consommatrice. La période de l'émetteur est alors nécessairement plus grande que la période de la tâche du *middleware* chargée de transmettre cette donnée. L'âge d'une occurrence de donnée est estimée comme étant la somme de la période de l'émetteur de la donnée et du pire temps de traversée de cette donnée pour rejoindre la tâche réceptrice.

Cette notion de fraîcheur est comparable à celle que nous avons décrite dans l'étude de cas (cf. sous-section 1.2.3 page 13) mise à part que nous considérons le cas où les données peuvent être écrasées, et donc ce n'est pas la consommation de l'occurrence suivante qui détermine l'âge maximal d'une occurrence de donnée, mais la consommation d'une nouvelle occurrence.

2.4.3 Evaluation de cohérence

Dans [90, 89], Nadège Pontisso propose une méthode pour analyser des propriétés de cohérence dans les systèmes distribués. Les résultats de cette approche sont fondés sur un formalisme prenant en compte les relations d'*in uence* entre les événements du système, ce qui permet de définir rigoureusement les notions abordées.

Principes. Un exemple de système étudié est représenté sur la figure 2.28 tirée de [89].

Le système est composé d'un ensemble de composants $C_1, \ldots, C_7$. Les flèches représentent les échanges de données entre ces composants. Cette méthode permet d'évaluer la cohérence d'association de données le long de *fuseau*. Un fuseau est un ensemble de chemins entre deux composants. L'idée est de vérifier que toutes les données utilisées par un composant en sortie d'un fuseau ont été produites lors d'une même exécution du composant en entrée du fuseau. Par exemple, toutes les données utilisées par C_7 ont pour origine C_1. L'association de données réalisée par C_7 est dite cohérente, si à chaque exécution de C_7, les données utilisées ont été produites lors de la même exécution de C_1.

Le modèle considéré est très générique. En particulier, aucune hypothèse n'est faite sur l'architecture sur laquelle s'exécutent les composants. Chaque composant C s'exécute avec une période T_C. Au démarrage de chaque période, un composant lit ses entrées et produit ses sorties avant le démarrage de

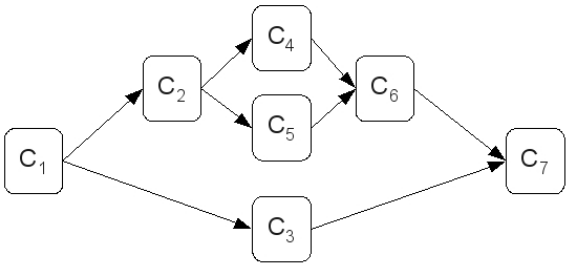


FIGURE 2.28 – Exemple de système étudié

la période suivante. Un composant est également caractérisé par un temps minimum d'exécution e_C. Les communications entre composants sont abstraites par des canaux de communication FIFO caractérisés par un temps minimum et maximum de transmission. Pour un canal entre les composants C et C , le temps minimum est noté $\delta_{CC'}$ et le temps maximum est noté $\Delta_{CC'}$.

Pour vérifier qu'une association de données est cohérente, la méthode consiste à mesurer l'écart maximal des latences sur les différents chemins d'un fuseau. En ce sens, cette méthode est proche de notre objectif de vérification de cohérence sur chaînes fonctionnelles divergentes. Pour deux chemins d'un fuseau tel que $P_A = (C_\alpha, C_2, \ldots, C_{n-1}, C_\beta)$ et $P_B = (C_\alpha, C_2, \ldots, C_{m-1}, C_\beta)$ de longueur respective n et m, un majorant de l'écart des latences est défini comme la différence entre la pire latence d'une chaîne et la meilleur latence de l'autre chaîne, ce qui donne :

$$ecart_{AB} = t_{\max}(P_A) + T_\beta - e_\beta - t_{\min}(P_B) \tag{2.23}$$

avec $t_{\max}(P)$ la latence maximale d'une chaîne $P = (C_1, \ldots, C_n)$, définie par :

$$t_{\max}(P) = \sum_{i=1}^{n-1} (2T_{C_i} + \Delta_{C_i C_{i+1}}) \tag{2.24}$$

et $t_{\min}(P)$ la latence minimale d'une chaîne $P = (C_1, \ldots, C_n)$, définie par :

$$t_{\min}(P) = \sum_{i=1}^{n-1} (e_{C_i} + \delta_{C_i C_{i+1}}) \tag{2.25}$$

Naturellement, pour déterminer l'écart pire cas entre les deux chemins, il faut également évaluer $ecart_{BA}$ qui est l'écart entre la latence maximale de P_B et la latence minimale de P_A.

Conclusion. Bien que ne reposant pas sur les hypothèses d'un système IMA, la méthode proposée est intéressante dans la mesure où elle formalise la notion de cohérence et propose un test simple pour son évaluation. Cependant, nous avons vu dans la présentation du *Real-Time Calculus* que ne pas considérer la corrélation entre le démarrage de certaines fonctions pouvait mener à une sur-approximation de la propriété évaluée. Or, notre objectif est justement de tirer parti des spécificités des systèmes IMA, pour obtenir des évaluations aussi précises que possibles.

D'autres travaux s'intéressent également à la notion de cohérence temporelle, par exemple les auteurs de [60] identifient ce besoin dans le domaine des bases de données temps réel. Cependant, l'évaluation de la cohérence proposée ne s'applique pas à des chaînes fonctionnelles. Les auteurs de [101] s'intéressent également à cette notion dans le contexte des systèmes distribués mais les techniques d'évaluation proposées reposent sur de la simulation et ne fournissent donc pas des garanties sûres.

2.5 Conclusion

Le problème posé concerne la vérification d'exigences temps réel appliquées aux systèmes IMA. Dans un premier temps, nous avons considéré l'utilisation d'un langage de description d'architecture,

tel qu'AADL, pour capturer l'architecture matérielle et logicielle d'un système IMA. De ce langage, nous conserverons la décomposition d'un système suivant trois vues : architecture fonctionnelle, architecture matérielle et allocation. Nous définirons au chapitre 4 une description formelle de l'architecture du système, n'incluant que les composants dont nous avons besoin.

Ensuite, pour décrire le comportement du système et ainsi permettre son analyse d'un point de vue temporel, nous nous sommes intéressés à des modèles généraux tels que les *automates temporisés*. Des exigences temps réel sont vérifiables sur ces modèles par *model checking*. Cependant, les spécificités des systèmes étudiés, en particulier l'asynchronisme et les différences d'échelles temporelles qui existent entre les composants, limitent l'utilisation de cette approche à des systèmes de tailles réduites. Nous avons alors fait le choix de modéliser le comportement du système dans le *tagged signal model*. Avec ce formalisme, il est possible de caractériser le comportement d'un système à l'aide de contraintes entre les événements du système. Il est alors envisageable d'utiliser des outils standards de propagation de contraintes ou de programmation linéaire pour analyser le modèle. L'exploitation de cette idée sera faite dans la troisième partie du manuscrit.

D'une manière générale, lors de l'analyse d'un système distribué, le réseau n'est pris en compte qu'au travers des performances qu'il offre, comme par exemple les temps de traversée meilleur et pire cas. C'est également la démarche que nous suivrons. Nous nous sommes par conséquent intéressés à deux techniques applicables aux réseaux AFDX : le *Network Calculus* et l'*approche par trajectoire*. Nous avons présenté les deux techniques car du point de vue de la précision des résultats fournis, on ne peut pas les départager.

Nous nous sommes également penchés sur les méthodes dédiées à l'analyse de latence, de fraîcheur ou de cohérence dans les systèmes distribués. Le *Real-Time Calculus*, par exemple, propose une modélisation dédiée à l'analyse de l'utilisation des ressources d'un système. Outre le pessimisme inhérent au modèle, cette méthode ne permet de vérifier que des exigences de latence. Ce qui ne répond que partiellement au problème posé. Un de nos objectifs est de proposer un cadre formel dans lequel tout type d'exigence temps réel pourrait être vérifié. Parmi, les travaux sur la cohérence dans les systèmes distribués, ceux de Nadège Pontisso ont retenu notre attention. Mais le modèle de système considéré ne correspond pas aux hypothèses des systèmes IMA et nous nous intéressons également à d'autres notions de cohérence qui ne sont pas définies dans ces travaux. Cependant, nous nous inspirerons de ces travaux pour définir des formules simples de cohérence (c.f. sous-section C page 165) que nous comparerons avec les résultats de notre méthode.

Chapitre 3

Contribution

Cette thèse est financée dans le cadre du projet ANR SATRIMMAP, composé de six partenaires : Airbus, CEA-LIST, IRIT, LAAS-CNRS, ONERA et QoS Design. Dans ce chapitre, nous présentons la démarche suivie ainsi que notre contribution dans la résolution du problème posé. Nous introduisons également l'outil de calcul que nous avons développé pour mettre en œuvre cette démarche, ainsi que son intégration dans le projet SATRIMMAP.

3.1 Démarche suivie

La figure 3.1 synthétise cette démarche, ainsi que le formalisme utilisé à chaque étape. Les données en entrée sont la description d'un système IMA ainsi que les exigences temps réel que le système doit satisfaire.

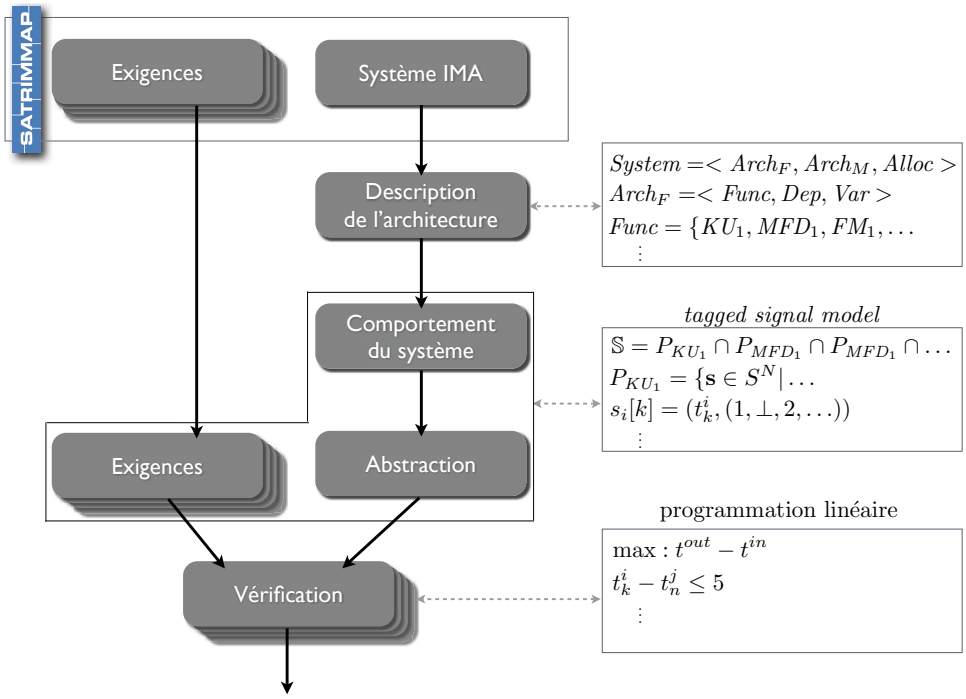


FIGURE 3.1 – Démarche suivie

3.1.1 Description de l'architecture

La première étape consiste à capturer les composants du système nécessaires à la définition du problème. Nous introduisons dans le chapitre 4 une formalisation de l'architecture des systèmes IMA. Le

modèle proposé se concentre uniquement sur les composants essentiels à la définition du problème. Ce modèle est donc beaucoup moins général qu'un modèle décrit avec un langage tel qu'AADL mais il gagne en simplicité. Un système est représenté selon trois vues : architecture fonctionnelle, architecture matérielle et relation d'allocation qui les lie. Des propriétés de correction sont définies dans ce formalisme. Elles permettent de contrôler, en amont de la vérification des exigences temps réel, que le système a du sens et donc que la vérification a un intérêt. Une telle propriété peut être, par exemple, « *soit f et f deux fonctions telles que f transmet une variable v à f et telles que les deux fonctions s'exécutent sur deux modules distincts m et m , alors il doit nécessairement exister un chemin dans le réseau entre m et m* ».

3.1.2 Formalisation des exigences et du comportement du système

La seconde étape consiste à décrire les comportements possibles des composants du système définis précédemment, ainsi que les comportements du système complet, c'est-à-dire les comportements résultant de l'assemblage de ces composants. Dans le chapitre 5, nous formalisons ces comportements à l'aide du *tagged signal model*, chaque élément du système étant modélisé par un processus. Pour un processus donné, nous exprimons les comportements acceptables comme les comportements dont tous les événements satisfont un ensemble de contraintes. Les comportements du système global sont alors obtenus par composition des processus.

Les exigences temps réel à vérifier sont exprimées dans le même formalisme au chapitre 6.

3.1.3 Abstraction

Lors de l'étape précédente, nous avons défini le comportement du système complet, mais en pratique la prise en compte de l'ensemble des composants du système induit un niveau de détail et une complexité qui sont incompatibles avec l'analyse d'un système industriel. La troisième étape consiste alors à définir une simplification sur-approximative des comportements d'un système IMA. La principale source de complexité du modèle provient de la modélisation des lisseurs de trafic des liens virtuels et des files d'attente du réseau. Ainsi, deux simplifications sont proposées. La première consiste à considérer qu'un lisseur de trafic est toujours utilisé au maximum de ses capacités, indépendamment du comportement de la fonction auquel il est rattaché. La deuxième consiste à abstraire le réseau par un ensemble de canaux temporisés. Chaque chemin de chaque lien virtuel est alors représenté par un canal, caractérisé par un intervalle de temps $[a, b]$: a et b étant respectivement les bornes inférieure et supérieure du temps de traversé du réseau le long du chemin. Ces bornes peuvent être déterminées avec le *Network Calculus* ou l'*approche par trajectoire*. Ces abstractions sont présentées au chapitre 7.

3.1.4 Vérification

La dernière étape consiste alors à vérifier les exigences temps réel sur le modèle abstrait des comportements. Il s'agit du cœur de notre contribution. Le *tagged signal model* est un formalisme très général et à ce titre, il n'existe pas d'outil qui lui soit propre. Nous avons alors fait le choix de traduire le problème de vérification d'une exigence en la résolution d'un programme linéaire en nombres entiers. Le pire, ou le meilleur, cas d'une exigence est alors exprimé comme la solution optimale d'un programme linéaire mixte, c'est-à-dire contenant des variables entières et réelles, dont les contraintes sont déduites de la modélisation dans le *tagged signal model*. Il est alors possible d'obtenir cette valeur avec un solveur comme `lp_solve` [24], qui est à la fois un solveur de programmes linéaires capable de résoudre des programmes linéaires (à variables réelles), des programmes linéaires en nombres entiers et des programmes linéaires mixte. Ces méthodes sont présentées au chapitre 8.

3.1.5 Publications

Les principaux résultats de l'approche décrite dans cette thèse ont été publiés dans les actes de deux conférences : l'article [70] regroupe les résultats concernant les exigences de latence et de fraîcheur et l'article [71] ceux concernant les exigences de cohérence sur chaînes fonctionnelles divergentes et convergentes.

La démarche proposée a également été présentée dans les actes de la conférences francophone MSR'11 (Modélisation des Systèmes Réactifs) [69], ainsi que lors des journées FAC'2011 (Formalisation des Activités Concurrentes).

Une première approche du problème, fondée sur une modélisation avec des automates temporisés, a été présentée dans [72]. Cet article propose également une modification du modèle d'entrée de l'*approche par trajectoire*. L'objectif étant de tirer partie des spécificités des systèmes IMA afin d'affiner les calculs de temps de traversée du réseau AFDX.

Un travail prospectif [68] sur l'évaluation de performances dans les réseaux Ethernet commutés a été publié dans les actes du *Junior Workshop* de la conférence RTNS'09 (*Real-Time and Network System*). Ce travail a également donné lieu à une présentation lors de l'école d'été RESCOM'09.

3.2 Développements réalisés

Un outil de calcul, fondé sur les travaux de cette thèse, a été développé. Il permet de vérifier qu'un système IMA satisfait un ensemble d'exigences temps réel, en évaluant les pires ou meilleurs cas des exigences. Dans cette section, nous donnons une présentation générale de cet outil, ainsi que son intégration dans le projet SATRIMMAP. L'outil sera détaillé à la section 7.3.

3.2.1 Un outil de vérification d'exigences temps réel

La figure 3.2 synthétise le fonctionnement de notre outil. Les développements ont été réalisés en *java* à l'aide de l'atelier de génie logiciel `Eclipse`. Nous utilisons deux librairies *open source* :

- `runCC` [49] qui permet de faciliter l'interprétation du fichier d'entrée contenant la description du système,
- `lp_solve` [24] qui est un solveur de programmes linéaires.

Les grandes étapes de l'analyse d'un système IMA sont les suivantes :

1. *Acquisition de la description du système et des exigences.* Un langage spécifique a été défini pour décrire un système et ses exigences. Cette première étape consiste donc en l'interprétation d'un fichier d'entrée, contenant la description d'un système, pour générer l'ensemble des objets qui le représentent. A l'issue de cette acquisition, un ensemble de tests est réalisé pour vérifier que le système est structurellement correct.
2. *Analyse du réseau* pour déterminer les paramètres des canaux temporisés. Nous avons développé un outil d'analyse réseau fondé sur l'*approche par trajectoire*. Il permet de déterminer les meilleurs et pires temps de traversée de chaque chemin du réseau.
3. *Pour chaque exigence, l'outil génère un MILP* (pour *Mixed Integer Linear Program*) dont la solution optimale correspond au pire ou au meilleur cas de l'exigence.
4. *La résolution de chaque MILP* est confiée à la librairie `lp_solve`, pour déterminer le pire ou meilleur cas de chaque exigence.
5. Finalement, on effectue une *comparaison* avec la borne acceptable de chaque exigence pour déterminer si l'exigence est satisfaite.

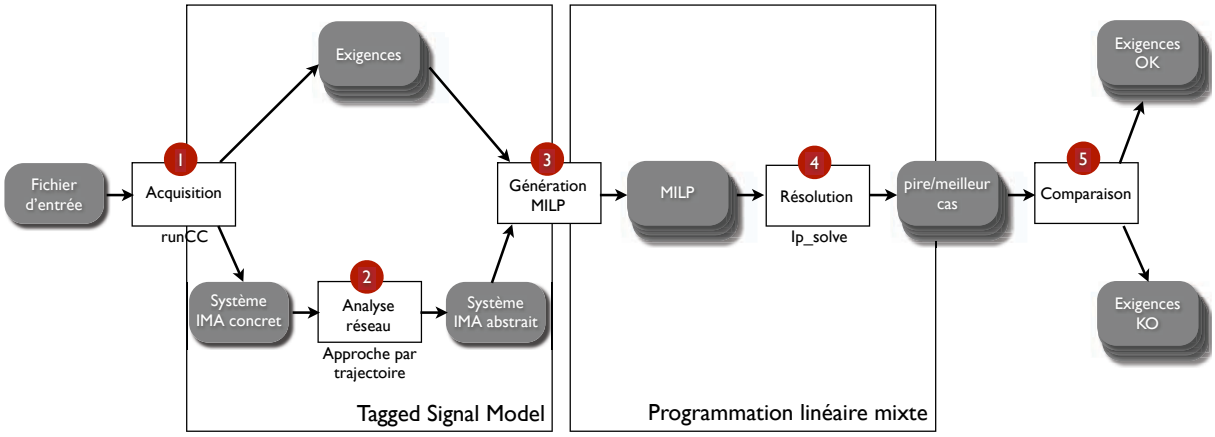


FIGURE 3.2 – Principes de l'outil de vérification

3.2.2 Intégration dans le projet SATRIMMAP

L'objectif principal du projet ANR SATRIMMAP est d'étudier et de définir un intergiciel permettant de faciliter la conception, le déploiement et la configuration d'applications distribuées sur une architecture IMA. L'ambition du projet n'est pas seulement d'étudier des services embarqués de type « middleware » sur une architecture temps réel, mais aussi et surtout d'étudier et développer des méthodes et outils pour faciliter le déploiement et la configuration de systèmes embarqués. Ce dernier point repose ainsi sur un ensemble d'outils *hors ligne*, dont fait partie le prototype développé dans le cadre de cette thèse. Le processus de conception SATRIMMAP, illustré sur la figure 3.3, est principalement composé des étapes suivantes :

1. Au niveau le plus haut, un modèle de composants a été défini pour décrire l'architecture fonctionnelle du système et les exigences temps réel associées. Ce modèle se focalise sur les aspects « métier », c'est-à-dire sur la modélisation système, indépendamment d'une infrastructure technologique particulière.
2. Dans une deuxième étape, une plateforme d'exécution spécifique est considérée, en l'occurrence l'ARINC 653, spécification standard de systèmes temps réel utilisés dans l'IMA. La gestion du déploiement et de la configuration est construite autour de la définition d'un (méta-)modèle de configuration d'une plateforme IMA. Les instances correspondantes sont, pour une partie, directement générées à partir du modèle de composants de plus haut niveau.
3. Les modèles de configuration issus de la deuxième étape sont dits « logiques », c'est-à-dire qu'ils n'incluent pas les paramètres de dépendance entre les applications et l'architecture matérielle ; en d'autres termes, ce sont des configurations non allouées. Un calcul de déploiement est alors effectué en fonction des exigences applicatives (par exemple période d'une partition), des contraintes systèmes (par exemple ségrégation de deux partitions) et des contraintes de l'architecture matérielle (par exemple ressource mémoire disponible). L'outil de déploiement *Deployment Solver* a été réalisé dans le cadre des travaux de thèse d'Ahmad Al Sheikh [5]. Cette allocation est ensuite évaluée par une analyse pire cas des latences de bout en bout sur toutes les chaines fonctionnelles. Cette analyse est réalisée par notre outil, nommé ici *Latency Checker*. Nous avons par conséquent développé une interface spécifique avec l'outil de calcul de déploiement (*Deployment Solver*). Si l'allocation respecte l'ensemble des exigences de latence, elle est alors intégrée à la configuration.

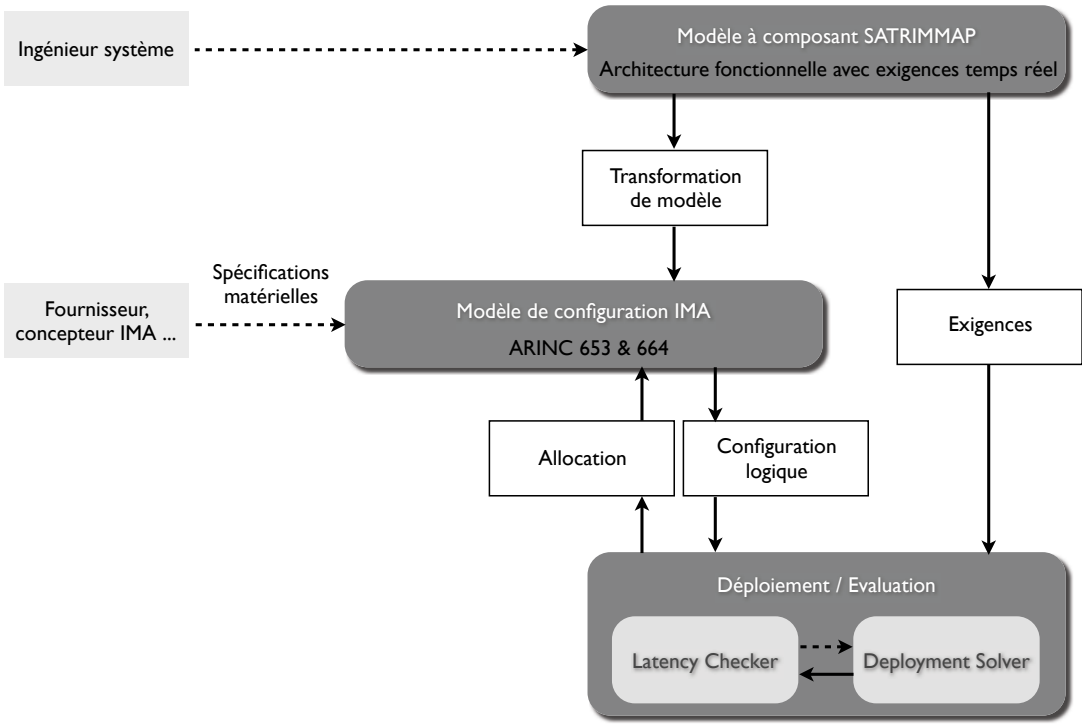


FIGURE 3.3 – Processus du projet SATRIMMAP

A noter que dans le projet SATRIMMAP, seules des exigences de latence sont considérées, mais notre outil est également capable de vérifier des exigences de fraîcheur et de cohérence entre chaînes fonctionnelles.

Deuxième partie

Modèle et sémantique

Introduction de la deuxième partie

Notre objectif est de vérifier que l'allocation d'un ensemble de fonctions avioniques sur une plateforme matérielle est satisfaisante du point de vue d'un ensemble d'exigences temporelles. Pour ce faire, l'objectif de cette partie est de donner une description formelle du problème.

Nous définissons d'abord le modèle *statique* du système (chapitre 4). Notre choix est de représenter les architectures fonctionnelle et matérielle séparément et ensuite de décrire la relation d'allocation qui les lie. Cette séparation en vues fonctionnelle, matérielle et allocation est assez standard. Ce découpage est notamment utilisé dans le langage de description d'architecture AADL [46]. On retrouve également cette idée dans les travaux sur l'allocation sûre de ressources avioniques de Laurent Sagaspe [95].

Ensuite, nous établissons un modèle *dynamique* du système représentant l'ensemble des comportements possibles du système (chapitre 5). Pour cela, nous modélisons dans le *tagged signal model* les composants élémentaires du système, dont le comportement est connu, et nous précisons la méthode pour exprimer le comportement du modèle global du système (c'est-à-dire de la plateforme opérationnelle composée des modèles d'architectures fonctionnelle, matérielle et d'allocation) à l'aide de ces composants. La notion de comportement du système est équivalente aux notions de trajectoire [15, 86], de trace ou d'exécution.

Finalement (chapitre 6), nous exprimons les exigences temporelles dans le modèle dynamique, ce qui fournira les fondations de notre méthode de vérification.

Chapitre 4

Modèle d architecture d un système IMA

L'objectif de ce chapitre est de décrire formellement l'architecture d'un système IMA. Un système S est composé de trois éléments :

$$S =< Arch_F, Arch_M, Alloc > \tag{4.1}$$

avec :
- $Arch_F$: la description de l'architecture fonctionnelle du système,
- $Arch_M$: la description de l'architecture matérielle du système,
- $Alloc$: l'allocation de l'architecture fonctionnelle sur l'architecture matérielle.

4.1 Architecture fonctionnelle

L'architecture fonctionnelle est composée d'un ensemble de fonctions *Func*, d'un ensemble de variables échangées entre ces fonctions *Var* et d'un ensemble de dépendances entre ces variables *Dep*. Ainsi, une architecture fonctionnelle est telle que :

$$Arch_F =< Var, Func, Dep > \tag{4.2}$$

avec

***Var* :** l'ensemble des variables échangées entre les fonctions. Une variable v *Var* est caractérisée par sa taille $v.size$ $\mathbb{N}$. La taille des variables est supposée constante.

On introduit une variable particulière représentant l'environnement extérieur, notée . Ceci permet d'identifier les fonctions interagissant avec l'environnement.

***Func* :** l'ensemble des fonctions. Une fonction f *Func* s'exécute périodiquement. A chaque exécution, une fonction lit un ensemble de variables en entrée, réalise un ensemble de traitements, et écrit le résultat des traitements sur un ensemble de variables en sortie. Les caractéristiques d'une fonction f *Func* sont alors :
- T_f $\mathbb{N}$ la période de la fonction,
- C_f $\mathbb{N}$ la durée maximale d'exécution de la fonction (également appelée WCET pour *Worst Case Execution Time*),
- $f.Input \subseteq Var$ l'ensemble des variables en entrée,
- $f.Output \subseteq Var$ l'ensemble des variables en sortie.

L'ensemble des fonctions est tel que :

$$Func \subseteq \mathbb{N} \times \mathbb{N} \times 2^{Var} \times 2^{Var} \tag{4.3}$$

Dans notre travail, nous ne nous intéressons pas au détail des différents traitements opérés au sein d'une fonction.

***Dep* :** l'ensemble des dépendances entre les variables. Toutes les sorties d'une fonction ne dépendent pas nécessairement de toutes ses entrées. Les dépendances des données du gestionnaire de vol (FM_1) de l'étude de cas sont représentées sur la figure 4.1 : la variable contenant les informations du *waypoint* ($wpInfo_1$) dépend uniquement de la réponse fournie par la base de données de navigation ($answser_1$). La variable de l'heure estimée d'arrivée au prochain *waypoint* (ETA_1) est calculée à partir des informations du *waypoint* et de la vitesse de l'appareil.

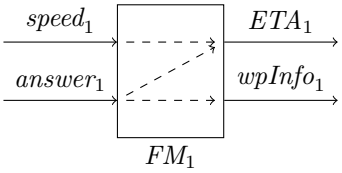


FIGURE 4.1 – Dépendances des variables de la fonction *Flight Manager*

On ne peut donc pas déduire les dépendances entre variables avec la seule connaissance des entrées et des sorties des fonctions. C'est pourquoi nous explicitons dans notre modèle les dépendances entre les variables avec l'ensemble *Dep*. Ainsi :

$$Dep \subseteq Var \times Var \tag{4.4}$$

Dans ce formalisme, les dépendances des variables de FM_1 s'expriment par :
- $(speed_1, ETA_1)$,
- $(answer_1, ETA_1)$,
- $(answer_1, wpInfo_1)$.

Une condition nécessaire pour qu'une variable v dépende directement d'une variable u est qu'il existe une fonction f telle que u soit en entrée de f et v en sortie. En revanche, comme toutes les sorties ne dépendent pas de toutes les entrées, ce n'est pas une condition suffisante. Plus formellement,

$$(u, v) \quad Dep \; = \quad f \quad Func \text{ tel que } u \quad f.Input \text{ et } v \quad f.Ouput \tag{4.5}$$

On dira également que u influence directement v.

$G = (Var, Dep)$ forme un graphe orienté appelé graphe de dépendances (non nécessairement connexe, pouvant comporter des cycles).

Nature d une variable. Une fonction s'exécute périodiquement mais elle ne produit pas nécessairement de nouvelles valeurs pour ses variables en sortie. Cela dépend de la nature périodique ou sporadique des variables. Sur l'exemple de la fonction FM_1 de la figure 4.1 la variable ETA_1 doit être mise à jour périodiquement pour prendre en compte en temps réel la progression de l'appareil. La nature de ETA_1 est donc périodique et les valeurs utilisées sont les dernières reçues sur les variables dont ETA_1 dépend. En revanche, la variable $wpInfo_1$ n'a besoin d'être actualisée que lorsque le pilote choisit un autre *waypoint*. Ainsi, la variable $wpInfo_1$ est sporadique : elle n'est produite que si une nouvelle valeur de $answer_1$ est disponible au démarrage de la fonction. Autrement dit, sa production est déclenchée par un événement sur la variable dont elle dépend.

Pour préciser cette idée, un exemple d'une séquence d'exécution de FM_1 est représenté sur la figure 4.2. Les flèches dirigées vers la ligne des temps sont les actualisations des variables en entrée de la fonction, celles s'en éloignant sont les écritures des variables en sortie à chacune de ses exécutions. Pour la variable ETA_1 les dépendances avec $speed_1$ et $answer_1$ sont précisées avec une valeur $(i, (j, k))$ où i est l'indice d'une occurrence ETA_1, j l'indice de l'occurrence de $speed_1$ utilisée dans le traitement et k celui de $answer_1$. Une nouvelle occurrence de $wpInfo_1$ est produite pour chaque réception de $answer_1$. Aucune nouvelle occurrence n'est reçue entre la troisième et la quatrième exécution de FM_1. Ainsi, seule ETA_1 est produite en utilisant les dernières occurrences reçues des variables $speed_1$ et $answer_1$.

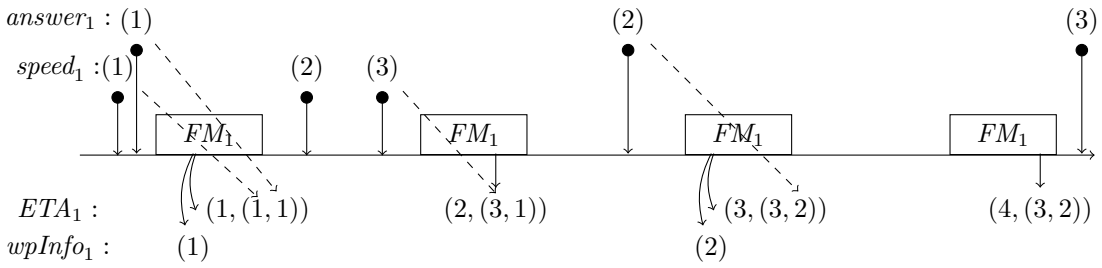


FIGURE 4.2 – Exemple de variables périodique (ETA_1) et sporadique ($wpInfo_1$)

Notons que la dépendance entre variables n'influence pas leur nature sporadique ou périodique. En d'autres termes, une variable « périodique » peut dépendre de variables « sporadiques » et une variable « sporadique » peut dépendre de variables « périodiques ».

Dans le cas où une variable de nature sporadique dépend de plusieurs variables, nous faisons le choix que chaque variable d'entrée peut être le déclencheur d'une production en sortie. Prenons l'exemple d'une fonction f avec deux variables u_1 et u_2 en entrée et une variable v de nature sporadique en sortie, dépendant de u_1 et u_2. Un comportement possible de f est représenté sur la figure 4.3.

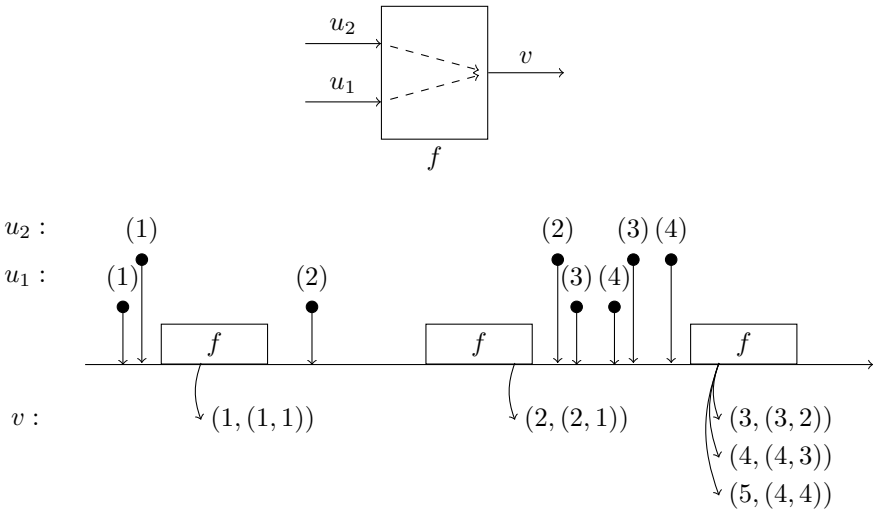


FIGURE 4.3 – Exemple d'une variable sporadique dépendant de deux variables

Lors de la première exécution de f, la dernière valeur reçue sur u_1 et celle sur u_2 sont appairées pour produire une nouvelle valeur de v. Pour la deuxième exécution de f, une nouvelle valeur est reçue sur au moins une des entrées (ici, sur u_1), ce qui déclenche une nouvelle production sur v. Comme il n'y a pas eu de nouvelle réception sur u_2, c'est sa dernière valeur qui est utilisée pour v. La troisième exécution montre comment les occurrences des variables en entrée sont appairées. Les valeurs reçues sont placées dans des files d'attente à politique de service « premier arrivé premier servi » (FIFO). Il y a une file par variable en entrée. Les variables sont alors appairées dans leur ordre d'arrivée. Ici, les occurrences 3 et 4 sont arrivées sur u_1 et les occurrences 2, 3 et 4 sur u_2. La première occurrence de v produite lors de cette exécution utilise donc l'occurrence 3 de u_1 et 2 de u_2, la deuxième utilise les occurrences 4 et 3. Pour la troisième, il n'y a plus de nouvelle arrivée sur u_1, mais il y en a une sur u_2. Ce sont alors l'occurrence la plus récente de u_1 (4) et la dernière arrivée de u_2 (4) qui sont utilisées.

Dé nition 1 (Nature d'une variable)**.** *Pour une variable v Var, on note $v.nat$ sa nature avec :*

$$v.nat \quad periodic,\ sporadic \tag{4.6}$$

Dé nition 2 (Architecture fonctionnelle bien formée)**.** *Une architecture fonctionnelle $Arch_F$ est dite bien formée si et seulement si :*

une variable n'est produite que par une unique fonction :

$$v \quad Var, \ ! \ f \quad Func \text{ telle que } v \quad f.Output \tag{4.7}$$

toute dépendance entre variables est réalisée par une fonction :

$$(u, v) \quad Dep, \ f \quad Func \text{ telle que } u \quad f.Input \text{ et } v \quad f.Output \tag{4.8}$$

Exemple 2 (Architecture fonctionnelle de l'étude de cas FMS)**.** *L'ensemble des fonctions de l'étude de cas est :*

$$Func = \ ADIRU_1, ADIRU_2, FM_1, FM_2, KU_1, KU_2, MFD_1, MFD_2, NDB \tag{4.9}$$

Les caractéristiques de ces fonctions sont résumées dans le tableau 4.1.

Fonction	Période (ms)	WCET (ms)	Entrée	Sortie
$ADIRU_1$	50	25	$pres_1$	$speed_1$
$ADIRU_2$	50	25	$pres_2$	$speed_2$
FM_1	60	30	$wpId_{1,2}, answer_1, speed_{1,2}$	$wpInfo_1, ETA_1$
FM_2	60	30	$wpId_{1,2}, answer_2, speed_{1,2}$	$wpInfo_2, ETA_2$
KU_1	50	25	req_1	$wpId_1$
KU_2	50	25	req_2	$wpId_2$
MFD_1	50	25	ETA_1	$disp_1$
MFD_2	50	25	ETA_2	$disp_2$
NDB	100	20	$query_1, query_2$	$answer_1, answer_2$

TABLE 4.1 – Caractéristiques des fonctions de l'étude de cas

L'ensemble des variables et des dépendances sont représentées sur le graphe 4.4. La taille et la nature des variables sont précisées dans le tableau 4.2.

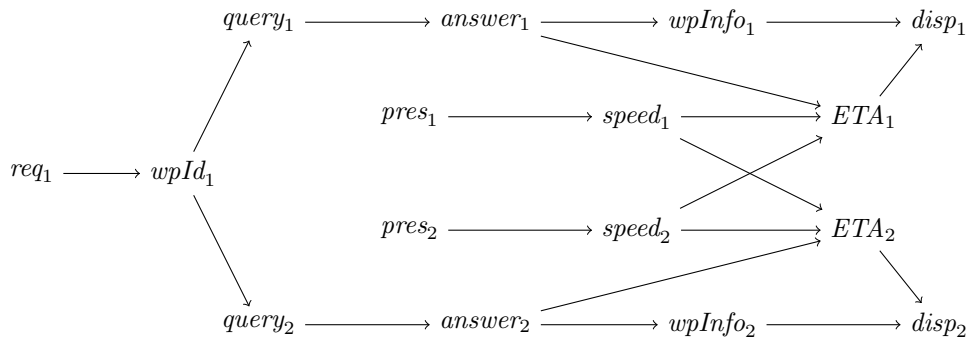


FIGURE 4.4 – Graphe de dépendance des variables

4.2 Architecture matérielle

Dans cette section, nous définissons l'architecture matérielle d'un système IMA. L'architecture matérielle est composée de l'ensemble des équipements mis en œuvre dans le système. Nous considérons les modules qui sont des ressources de calcul, les commutateurs et les liens du réseau AFDX qui forment les ressources de communication, les capteurs, les actionneurs et les RDC. Un capteur (ou un actionneur) peut être directement relié à un module via un bus de terrain, ou alors peut être relié à un module au travers du réseau AFDX. Dans ce cas de figure, un RDC est utilisé pour assurer l'interconnexion entre le bus du capteur (ou de l'actionneur) et le réseau AFDX.

Une architecture matérielle est telle que :

Variable	Taille	Nature
req_1, req_2	600	sporadic
$wpId_1$, $wpId_2$	600	sporadic
$query_1$, $query_2$	1000	sporadic
$answer_1$, $answer_2$	4000	sporadic
$wpInfo_1$, $wpInfo_2$	5000	sporadic
$pres_1$, $pres_2$	512	periodic
$speed_1$, $speed_2$	800	periodic
ETA_1, ETA_2	1000	periodic
$disp_1$, $disp_2$	8000	periodic

TABLE 4.2 – Taille et nature des variables du FMS

$$Arch_M =< Ports, Modules, Switches, Links, RDC, Sensors, Actuators > \tag{4.10}$$

avec

Ports **:** l'ensemble des ports assurant la connexion au réseau. Les équipements possédant des ports sont les modules, les RDC et les commutateurs. Un port est bi-directionnel : dans un sens il est connecté à un équipement, dans l'autre il est relié à un lien connecté au reste du réseau. Le port est responsable du multiplexage du trafic réseau produit par l'équipement et envoyé sur le lien. Il dispose donc d'une file d'attente pour gérer les accès concurrents au lien. On considère que le port est suffisamment rapide pour démultiplexer le trafic arrivant sur le lien au fur et à mesure qu'il arrive. Il n'y a donc pas besoin de file d'attente pour gérer le démultiplexage du trafic arrivant depuis le lien. Cela suppose donc que deux ports reliés par un même lien sont caractérisés par un même débit maximal. Pour un module p $Ports$, on note $p.d$ $\mathbb{N}$ son débit.

Modules **:** l'ensemble des modules. Un module est composé d'un CPU et d'un port de communication interfacé avec le réseau AFDX. Pour un module M $Modules$, on note $M.port$ $Ports$ son port AFDX.

RDC **:** l'ensemble des RDC (pour *Remote Data Concentrator*). Un RDC est utilisé pour interconnecter un ensemble de capteurs/actionneurs avec le réseau AFDX. Ce type de composant est constitué de deux fonctions exécutées périodiquement : encapsulation dans des trames AFDX des variables fournies par les capteurs et désencapsulation des variables contenues dans les trames AFDX à destination des actionneurs. Pour un RDC r RDC, on note :

- T_r $\mathbb{N}$ sa période,
- C_r $\mathbb{N}$ son temps de traitement maximal, c'est-à-dire le temps maximal requis pour réaliser l'ensemble des encapsulations et des désencapsulations,
- $r.port$ $Ports$ son port connecté au réseau AFDX.

Switches **:** l'ensemble des commutateurs. Un commutateur est un regroupement de ports de communication. Il est caractérisé par une latence technologique décrivant le temps de commutation entre deux ports (cette latence est estimée à 140 μs pour les commutateurs de l'A380). Pour un commutateur s $Switches$, on note :

- $s.ports \subseteq Ports$ l'ensemble des ports du commutateur,
- $s.lag$ $\mathbb{N}$ la latence technologique.

Links **:** l'ensemble des liens entre ports de communication. Les liens physiques utilisés sont bi-directionnels et servent à relier une paire de ports. On a alors $Links \subseteq Ports^2$. Soit deux ports p_1 et p_2, si il existe un lien entre p_1 et p_2 alors (p_1, p_2) $Links$ et (p_2, p_1) $Links$ représentent le même lien physique.

Un lien est tel qu'il relie un module à un commutateur, un RDC à un commutateur ou deux commutateurs entre eux. Plus formellement, un lien (p_1, p_2) $Links$ vérifie toujours :

- $p_1 = p_2$: un port ne peut être lié à lui même,

- m *Modules* tel que $p_1 = m.port$ et s *Switches* tel que p_2 $s.ports$,
- ou s *Switches* tel que p_1 $s.ports$ et m *Modules* tel que $p_2 = m.port$
- ou r *RDC* tel que $p_1 = r.port$ et s *Switches* tel que p_2 $s.ports$,
- ou s *Switches* tel que p_1 $s.ports$ et r *RDC* tel que $p_2 = r.port$
- ou (s_1, s_2) $Switches^2$ tel que p_1 $s_1.ports$ et p_2 $s_2.ports$ avec $s_1 = s_2$

***Sensors* :** l'ensemble des capteurs. Les capteurs permettent de transformer une mesure physique de l'environnement extérieur en une variable interprétable par le système. A l'image de la nature des variables manipulées par le système (périodique ou sporadique), un capteur peut fonctionner de deux façons : soit il échantillonne périodiquement une valeur physique (mesure de la pression extérieure à la carlingue par exemple), soit il réagit à un événement (dépassement d'un seuil, saisie d'une donnée par le pilote,...). Un capteur événementiel ne peut détecter l'ensemble des sollicitations extérieures : il ne peut se déclencher qu'un nombre fini de fois pendant une période de temps donnée. Nous définissons cette limitation par le temps minimum séparant la détection de deux événements successifs. En outre, cela permet d'empêcher qu'une infinité d'événements entre dans le système durant un temps fini.

Un capteur peut être connecté à un module directement ou alors être interconnecté au réseau au travers d'un RDC. Dans les deux cas, le capteur est relié à l'aide d'un bus de terrain tel qu'un bus CAN (pour *Controller Area Network*) [73] par exemple. Nous ne nous intéressons ni à la nature de ce lien ni à ses caractéristiques fines, nous supposerons seulement que son temps de traversée est compris dans un intervalle donné. La méthode proposée dans [106] peut être utilisée dans le cas d'un bus CAN pour dimensionner cet intervalle.

Pour un capteur s *Sensors* on note :
- $s.nat$ $periodic, sporadic$ sa nature périodique ou sporadique,
- T_s un temps représentant soit sa période d'échantillonnage, soit le temps minimum entre deux événements,
- $[a, b]$ un intervalle représentant le temps de traversée du bus de terrain le reliant au système,
- $s.equipement$ *Modules* *RDC* l'équipement auquel est rattaché le capteur via le bus de terrain.

***Actuators* :** Les actionneurs sont utilisés par le système pour interagir avec son environnement. Un actionneur transforme donc une variable en une valeur physique. On supposera qu'un actionneur produit une valeur physique à chaque changement de valeur de sa variable. Son comportement est donc sporadique. Tout comme un capteur, un actionneur est relié à un module ou un RDC au travers d'un bus de terrain. Pour un actionneur a *Actuators*, on note :
- $[a, b]$ un intervalle représentant le temps de traversée du bus,
- $a.equipement$ *Modules* *RDC* l'équipement auquel est rattaché l'actionneur via le bus de terrain.

Pour être bien formée, une architecture matérielle doit fournir une liaison entre chaque RDC et au moins un module. Pour préciser cette idée, nous définissons préalablement la notion de chemin dans le réseau.

Dé nition 3 (Chemin réseau)**.** *Soit une séquence de n liens du réseau $(l_1, \ldots, l_n)$ $Links^n$. $(l_1, \ldots, l_n)$ définit un chemin dans le réseau si et seulement si les liens sont interconnectés deux à deux dans des commutateurs. Autrement dit, chaque paire de liens consécutifs possède un port appartenant à un même commutateur : $i = 1..n-1$, s Switches t.q. $l_i \setminus s.ports =$ et $l_{i+1} \setminus s.ports =$.*

Nous nous intéressons à l'ensemble des chemins reliant les modules et les RDC, autrement dit les chemins de bout en bout du réseau AFDX. L'ensemble de ces chemins est noté EtePaths (pour End to end paths*) et est défini par :*

$$EtePath \subseteq 2^{Links} \tag{4.11}$$

Soit un chemin p EtePath. Nous notons $p.start$ son premier port et $p.end$ son dernier port.

Dé nition 4 (Architecture matérielle bien formée)**.** *Une architecture matérielle est dite* bien formée *si chaque RDC est lié à au moins un module :*

$$\begin{aligned} & R \quad RDC, \\ & p \quad EtePath \text{ avec } R.port = p.start \\ \text{et} \quad & M \quad Modules \text{ avec } M.port = p.end \end{aligned} \tag{4.12}$$

Exemple 3 (Architecture matérielle de l'étude de cas)**.** *L'architecture matérielle $Arch_M$ de l'étude de cas (cf. figure 1.4 page 10) est composée des éléments suivants :*

Un ensemble de 7 modules,

$$Modules = \ M_1, M_2, M_3, M_4, M_5, M_6, M_7 \tag{4.13}$$

avec $i = 1..7$, $M_i.1$ *le port du module* M_i.

Un ensemble de 5 commutateurs,

$$Switches = \ S_1, S_2, S_3, S_4, S_5 \tag{4.14}$$

avec $i = 1..5$, $S_i.lag = 140\mu s$ *et* $S_i.j$ *désigne le port* j *du commutateur* S_i.

Un ensemble de 2 Remote Data Concentrators,

$$RDC = \ R_1, R_2 \tag{4.15}$$

avec $R \quad RDC, T_R = 50ms$ *et* $C_R = 10ms$.

Un ensemble de 13 liens entre les ports des équipements,

$$\begin{aligned} Links = \ & (S_1.1, M_1.1), (S_1.2, M_7.1), (S_1.3, S_2.2), (S_1.4, S_3.2), \\ & (S_1.5, S_4.2), (S_1.6, S_5.2), (S_1.7, M_2.1), (S_2.1, M_3.1) \\ & (S_3.1, M_4.1), (S_4.1, M_5.1), (S_4.3, R_1.1), (S_5.1, M_6.1) \\ & (S_5.3, R_2.1) \end{aligned} \tag{4.16}$$

Un ensemble de 4 capteurs

$$Sensors = \ key_1, key_2, sensor_1, sensor_2 \tag{4.17}$$

dont les caractéristiques sont données dans le tableau suivant,

Capteur	*Nature*	*Période (ms)*	*Temps traversée bus (ms)*	*équipement*
key_1	*sporadic*	*60*	$[0.1, 0.2]$	M_1
key_2	*sporadic*	*60*	$[0.1, 0.2]$	M_2
$sensor_1$	*periodic*	*20*	$[0.1, 0.2]$	R_1
$sensor_2$	*periodic*	*20*	$[0.1, 0.2]$	R_2

Un ensemble de 2 actionneurs,

$$Actuators = \ display_1, display_2 \tag{4.18}$$

dont les caractéristiques sont données dans le tableau suivant,

Actionneur	*Temps traversée bus (ms)*	*équipement*
$display_1$	$[0.1, 0.2]$	M_1
$display_2$	$[0.1, 0.2]$	M_2

4.3 Allocation sur la plateforme

L'architecture fonctionnelle est projetée sur l'architecture matérielle à l'aide de quatre fonctions d'allocation. La première ($Alloc_F$) alloue chaque fonction à une partition d'un module, les deuxième et troisième allouent les variables aux capteurs et aux actionneurs suivant que ces variables sont en entrée ($Alloc_S$) ou en sortie ($Alloc_P$) du système, la quatrième ($Alloc_D$) alloue à chaque dépendance entre fonctions les ressources de communication entre ces fonctions. Ces ressources sont des liens virtuels (des VL pour *Virtual Link*). La figure 4.5 synthétise les quatre fonctions d'allocation proposées.

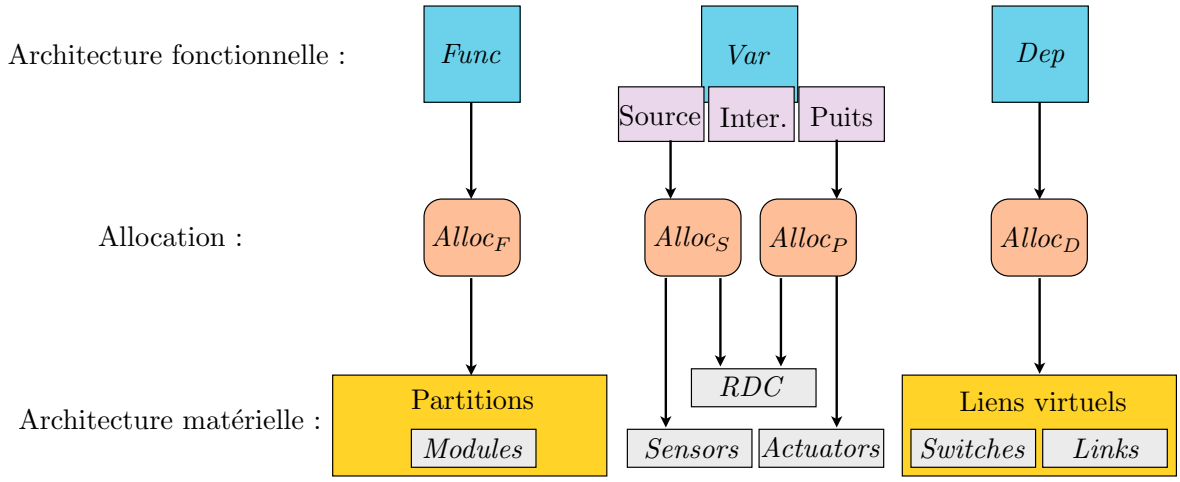


FIGURE 4.5 – Modèle d'allocation

4.3.1 Allocation des fonctions

Une fonction est allouée dans une partition qui est exécutée sur un module. Nous considérons qu'une unique fonction peut être allouée à une partition. Pour limiter la complexité du modèle, une partition est définie sur une unique tranche de temps. Notons que cette hypothèse correspond au manuel de « bonnes pratiques » défini chez Airbus, l'idée étant de limiter les arrêts/redémarrages des processus internes à une partition. Nous définissons dans un premier temps une partition.

Dé nition 5 (Partition)**.** *Une partition est rattachée à un module et est caractérisée par un ordonnancement statique sur ce module. Ainsi une partition est définie par les paramètres suivants :*

M *Modules : le module hébergeant la partition,*
T $\mathbb{N}$ *: la période de la partition,*
O $\mathbb{N}$ *: le décalage entre le démarrage de la partition et le démarrage de la MAF (ce décalage est également appelé l'*offset*),*
C $\mathbb{N}$ *: la durée de la partition.*

L'ensemble des partitions est $Partitions \subseteq Modules \times \mathbb{N}^3$.

Dé nition 6 (Allocation de fonctions)**.** *Une allocation des fonctions est définie à l'aide de la fonction $Alloc_F$ qui, pour chaque fonction, retourne une partition :*

$$Alloc_F : Func \quad Partitions \tag{4.19}$$

Ainsi, pour f $Func$, $Alloc_F(f) = p = (M, T, O, C)$ signifie que f s'exécute dans la partition p qui est hébergée par le module M et qui est ordonnancée avec une période T, un o set O et une durée C.

Dans le cadre du projet SATRIMMAP, Ahmad al Sheikh a défini dans ses travaux [6, 98] différentes méthodes d'allocation de l'architecture fonctionnelle sur l'architecture matérielle. Nous utilisons ses conditions de correction pour définir une allocation de fonction bien formée.

Dé nition 7 (Allocation de fonctions bien formée)**.** *Une allocation des fonctions est dite bien formée si elle satisfait les propriétés suivantes :*

non entrelacement des partitions (ou pas de superpositions) : à tout instant, au plus une partition s'exécute sur un module. Une condition nécessaire et su sante pour assurer cette propriété a été proposée par Korst dans [62]. En appliquant cette condition à notre modèle pour deux fonctions (f, g) $Func$ allouées sur un même module M $Modules$ et telles que $Alloc_F(f) = (M, T_1, O_1, C_1)$ et $Alloc_F(g) = (M, T_2, O_2, C_2)$, nous obtenons : f et g peuvent s'exécuter sur M si et seulement si

$$C_1 \leq [(O_2 - O_1) \mod pgcd(T_1, T_2)] \leq pgcd(T_1, T_2) - C_2 \tag{4.20}$$

la période de la fonction et celle de la partition doivent être identiques : f $Func$ telle que $Alloc_F(f) = (M_1, T_1, O_1, C_1)$, $T_f = T_1$

la durée de la partition doit être supérieure ou égale au pire temps d'exécution de la fonction : $C_f \leq C_1$

Dans la suite, les notions de période et de temps d'exécution des fonctions sont confondues avec les notions de période et de durée des partitions.

Exemple 4 (Allocation des fonctions de l'étude de cas FMS)**.** *Une allocation bien formée des fonctions de l'étude de cas est donnée par la fonction* $Alloc_F$ *telle que :*

$$\begin{aligned}
Alloc_F(KU_1) &= (M_1, 50, 0, 25)\\
Alloc_F(MFD_1) &= (M_1, 50, 25, 25)\\
Alloc_F(KU_2) &= (M_2, 50, 0, 25)\\
Alloc_F(MFD_2) &= (M_2, 50, 25, 25)\\
Alloc_F(FM_1) &= (M_3, 60, 0, 30)\\
Alloc_F(FM_2) &= (M_4, 60, 0, 30)\\
Alloc_F(ADIRU_1) &= (M_5, 60, 0, 30)\\
Alloc_F(ADIRU_2) &= (M_6, 60, 0, 30)\\
Alloc_F(NDB) &= (M_7, 100, 0, 20)
\end{aligned}$$

Cette allocation est bien formée.

4.3.2 Allocation des variables

Pour l'allocation, on distingue trois catégories de variables : les variables *sources* issues des capteurs, les variables *puits* contrôlant les actionneurs et les variables *intermédiaires* échangées entre les fonctions. Une variable source doit être allouée à un capteur et une variable puits à un actionneur. Les variables intermédiaires n'ont pas besoin d'autres ressources que celles déjà allouées à leurs fonctions respectives.

Les variables peuvent donc être regroupées en trois ensembles :

– l'ensemble des variables sources qui ne dépendent d'aucune autre variable :

$$Var_S = \quad v \quad Var \quad (\quad ,v) \quad Dep \tag{4.21}$$

– l'ensemble des variables puits qui influencent l'environnement :

$$Var_P = \quad u \quad Var \quad (u, \quad) \quad Dep \tag{4.22}$$

– l'ensemble des variables intermédiaires :

$$Var_I = \quad v \quad Var \quad (v_1, v_2) \quad Var^2 \text{ t.q. } (v_1, v) \quad Dep \quad (v, v_2) \quad Dep \tag{4.23}$$

Précisons que potentiellement les ensembles des variables puits et intermédiaires ne sont pas disjoints. En effet, une variable en sortie d'une fonction et à destination d'un actionneur peut également être utilisée en entrée d'une autre fonction.

On précise l'allocation des variables à l'aide de deux fonctions, $Alloc_S$ pour les variables sources et $Alloc_P$ pour les variables puits.

Dé nition 8 (Allocation des variables sources)**.** *La fonction d'allocation des variables sources associe à chaque variable source un capteur :*

$$Alloc_S : Var_S \quad Sensors \tag{4.24}$$

Dé nition 9 (Allocation des variables sources bien formée)**.** *Pour être bien formée, une allocation des variables sources doit respecter :*

un capteur ne produit qu'une unique variable et *une variable n'est produite que par un unique capteur :*

$$\begin{aligned}
&s \quad Sensors, \ !v \quad Var_S \ t.q. \ Alloc_S(v) = s\\
&v \quad Var_S, \ !s \quad Sensors \ t.q. \ Alloc_S(v) = s
\end{aligned} \tag{4.25}$$

si le capteur est directement relié à un module, alors toutes les fonctions utilisant la variable du capteur s'exécutent sur ce module :

$$\begin{array}{l} v \quad Var_S, \ M \quad Modules, \ f \quad Func \\ (Alloc_S(v).equipement = M \quad v \quad f.Input) \ = \quad Alloc_F(f) = (M, T_f, O_f, C_f) \end{array} \tag{4.26}$$

si le capteur est relié à un RDC, alors pour chaque module où s'exécute une fonction utilisant la variable, il existe un chemin entre le RDC et le module :

$$\begin{array}{l} v \quad Var_S, \ r \quad RDC, \ M \quad Modules, \ f \quad Func \\ (v \quad f.Input \quad Alloc_S(v).equipement = r \quad Alloc_F(f) = (M, T_f, O_f, C_f)) \\ = \quad p \quad EtePath \ t.q. \ r.port = p.start \quad M.port = p.end \end{array} \tag{4.27}$$

Nous définissons désormais la fonction d'allocation des variables puits.

Dé nition 10 (Allocation des variables puits)**.** *La fonction d'allocation des variables puits associe à chaque variable puits un actionneur :*

$$Alloc_P : Var_P \quad Actuators \tag{4.28}$$

Dé nition 11 (Allocation des variables puits bien formée)**.** *Pour être bien formée, une allocation des variables puits doit respecter :*
un actionneur ne réagit qu'à une unique variable et *une variable n'est envoyé qu'à un unique actionneur :*

$$\begin{array}{l} a \quad Actuators, \ !v \quad Var_P \ t.q. \ Alloc_P(v) = a \\ v \quad Var_P, \ !a \quad Actuators \ t.q. \ Alloc_P(v) = a \end{array} \tag{4.29}$$

si l'actionneur est directement relié à un module, la fonction productrice de v doit être allouée sur ce module :

$$\begin{array}{l} v \quad Var_P, \ M \quad Modules, \ f \quad Func \\ (Alloc_P(v).equipement = M \quad v \quad f.Output) \ = \quad Alloc_F(f) = (M, T_f, O_f, C_f) \end{array} \tag{4.30}$$

si l'actionneur est relié au travers d'un RDC, il existe un chemin entre le RDC et le module où est produite la variable :

$$\begin{array}{l} v \quad Var_P, \ r \quad RDC, \ M \quad Modules, \ f \quad Func \\ (v \quad f.Output \quad Alloc_P(v).equipement = r \quad Alloc_F(f) = (M, T_f, O_f, C_f)) \\ = \quad p \quad EtePath \ t.q. \ r.port = p.start \quad M.port = p.end \end{array} \tag{4.31}$$

Exemple 5 (Allocation des variables de l'étude de cas FMS)**.** *L'ensemble des variables sources de l'étude de cas est tel que :*

$$Var_S = \ req_1, req_2, pres_1, pres_2 \tag{4.32}$$

Une allocation bien formée des variables sources de l'étude de cas est donnée par la fonction $Alloc_S$ telle que :

$$\begin{array}{r} Alloc_S(req_1) = key_1 \\ Alloc_S(req_2) = key_2 \\ Alloc_S(pres_1) = sensor_1 \\ Alloc_S(pres_2) = sensor_2 \end{array}$$

L'ensemble des variables puits de l'étude de cas est tel que :

$$Var_P = \quad disp_1, disp_2 \tag{4.33}$$

Une allocation bien formée des variables puits de l'étude de cas est donnée par la fonction $Alloc_P$ telle que :

$$Alloc_P(disp_1) = display_1$$
$$Alloc_P(disp_2) = display_2$$

4.3.3 Allocation des dépendances

Chaque dépendance entre variables exprime un besoin de communication entre deux entités (fonction, capteur, actionneur ou RDC). La communication se fait localement si les deux entités sont localement reliées par un bus de terrain, ou si elles sont localisées sur le même module. Pour les autres cas, une variable est allouée dans un paquet qui transite sur un *lien virtuel* (ou VL pour *Virtual Link)*. Un lien virtuel est un lien logique multicast reliant une source et une ou plusieurs destinations. Il faut donc plusieurs chemins de l'ensemble *EtePath* pour décrire un chemin *multicast* dans le réseau. Nous formalisons la notion de lien virtuel :

Dé nition 12 (Lien virtuel)**.** *Un lien virtuel est défini par les tailles minimale et maximale des trames émises, l'ensemble des chemins qu'il couvre sur le réseau et le temps minimal entre deux émissions de trames (BAG). L'ensemble des liens virtuels est noté VL. Soit un lien virtuel vl VL, nous notons :*

$vl.s_{min}$ $\mathbb{N}$ la taille minimale d'une trame,
$vl.s_{max}$ $\mathbb{N}$ la taille maximale d'une trame,
$vl.paths$ $2^{EtePath}$ les chemins du VL,
$vl.bag$ $\mathbb{N}$ le temps minimal entre deux émissions de trames.

L'ensemble des liens virtuels est alors :

$$VL \subseteq \mathbb{N} \times \mathbb{N} \times 2^{EtePath} \times \mathbb{N} \tag{4.34}$$

Dé nition 13 (Allocation des dépendances)**.** *La fonction d'allocation des dépendances $Alloc_D$ retourne pour chaque dépendance un VL, ou pour les dépendances ne nécessitant pas de VL.*

$$Alloc_D : Dep \quad VL \tag{4.35}$$

Exemple 6 (Allocation des dépendances)**.** *Pour illustrer la notion d'allocation des dépendances, nous prenons l'exemple d'une dépendance de l'étude de cas : $(wpId_1, query_1)$. La variable $wpId_1$ est produite par la fonction KU_1 et est consommée par FM_1 pour produire $query_1$. La dépendance $(wpId_1, query_1)$ exprime donc un besoin de communication entre ces fonctions. Sur l'étude de cas, cette communication s'e ectue via le VL vl_1, ce qui s'exprime :*

$$Alloc_D(wpId_1, query_1) = vl_1 \tag{4.36}$$

avec :

$$\begin{aligned} &vl_1.s_{min} = 600 \\ &vl_1.s_{max} = 600 \\ &vl_1.paths = \quad (M_1.1, S_1.1), (S_1.3, S_2.2), (S_2.1, M_3.1) \ , \ (M_1.1, S_1.1), (S_1.4, S_3.2), (S_3.1, M_4.1) \\ &vl_1.bag = 4 \end{aligned} \tag{4.37}$$

Ce VL est composé de deux chemins. En e et, la variable $query_2$, produite par FM_2, dépend également de $wpId_1$ et cette dépendance est également allouée sur vl_1. Une allocation de l'ensemble des dépendances de l'étude de cas est donnée en exemple à la fin de cette section.

Pour faciliter la définition des propriétés de correction d'une allocation des dépendances, on définit la fonction *varVL* retournant l'ensemble des variables traversant un VL :

$$\begin{aligned} varVL : VL \quad & 2^{Var} \\ vl \quad & u \quad Var \quad (u, v) \quad Dep \quad Alloc_D(u, v) = vl \end{aligned} \tag{4.38}$$

On a alors, par exemple, $varVL(vl_1) = \quad wpId_1$.

Dé nition 14 (Allocation des dépendances bien formée)**.** *Une allocation des dépendances bien formée doit respecter :*

la taille minimale des trames d'un VL doit être inférieure ou égale à la taille des variables le traversant :

$$vl \quad VL \ , \ vl.s_{min} \leq \min_{v \quad varVL(vl)} v.size \tag{4.39}$$

la taille maximale des trames d'un VL doit être supérieure ou égale à la taille des variables le traversant :

$$vl \quad VL \ , \ vl.s_{max} \geq \max_{v \quad varVL(vl)} v.size \tag{4.40}$$

toutes les variables utilisant un VL sont produites par une même source (soit toutes les variables sont produites par une unique fonction, soit toutes les variables sont produites par des capteurs rattachés à un même et unique RDC) :

$$\begin{array}{lll} & vl & VL, \\ soit & !f & Func \ t.q. \ varVL(vl) \subseteq f.Output, \\ soit & !r & RDC \ t.q. \quad v \quad varVL(vl), Alloc_S(v).equipement = r \end{array} \tag{4.41}$$

pour chaque dépendance (u, v) Dep allouée à un VL et avec v consommée par une fonction, il existe au moins un chemin de ce VL rejoignant le module de cette fonction :

$$\begin{array}{l} (u, v) \quad Dep, \ vl \quad VL, \ f \quad Func, \ M \quad Modules \\ (Alloc_D(u, v) = vl \quad v \quad f.Input \quad Alloc_F(f) = (M, T_f, O_f, C_f)) \\ = \quad p \quad vl.paths \ t.q. \ M.port = p.end \end{array} \tag{4.42}$$

pour chaque dépendance (u, v) Dep allouée à un VL et avec v consommée par un RDC, il existe au moins un chemin de ce VL rejoignant ce RDC :

$$\begin{array}{l} (u, v) \quad Dep, \ vl \quad VL, \ r \quad RDC \\ (Alloc_D(u, v) = vl \quad Alloc_P(v).equipement = r) \ = \quad p \quad vl.paths \ t.q. \ r.port = p.end \end{array} \tag{4.43}$$

les chemins d'un VL forment un arbre (les chemins ont tous le même premier lien et ne se croisent pas une fois qu'ils se sont séparés) :

$$\begin{array}{l} vl \quad VL, \ (\ l_1, \ldots, l_m \ , \ l_1, \ldots, l_{m'} \ ) \quad vl.paths^2 \\ l_1 = l_1 \quad (\ i = 2..\min \ m, m \quad t.q. \ l_i = l_i) \ = \quad j \quad \mathbb{N} \ t.q. \ i < j \leq \min \ m, m \quad , \ l_j = l_j \end{array} \tag{4.44}$$

le graphe des chemins des VL est acyclique. L'algorithme de Tarjan [103] pour la recherche de composantes fortement connexes dans les graphes orientés peut être utilisé pour tester cette propriété, avec une complexité $O(E + V)$ (E le nombre d'arcs du graphe et V le nombre de n uds). Si il n'existe pas de composante fortement connexe non triviale (ayant plus d'un unique n ud) alors il n'existe pas de cycle dans le graphe.
un VL ne perd pas de trames. Les liens du réseau sont full duplex *ce qui signifie qu'il ne peut pas y avoir de collision de trames sur les liens. Une trame ne peut donc être perdue que si elle est réceptionnée par un composant dont la file d'attente est pleine. Pour garantir qu'aucune trame n'est perdue, il faut donc prouver que les capacités des files d'attente des lisseurs de trafic, ainsi que celles des commutateurs, ne sont jamais dépassées. Le* Network Calculus *[74] est classiquement utilisé pour ces preuves. Cette technique a été utilisée avec succès sur des réseaux AFDX dans [52, 48, 37, 31].*

Une allocation bien formée garantit donc qu'un lisseur de trafic a terminé de transmettre les trames produites lors d'une exécution, avant l'exécution suivante. En particulier, le concepteur de l'allocation

doit garantir que pour toute fonction f et tout lien virtuel vl associé à cette fonction, il existe un nombre maximal N_{vl} de trames émises lors d'une exécution, et N_{vl} doit respecter :

$$(N_{vl} - 1) \cdot bag \leq T_f - C_f \tag{4.45}$$

Cette condition garantit la stabilité du lisseur de trafic et elle est illustrée sur la figure 4.6.

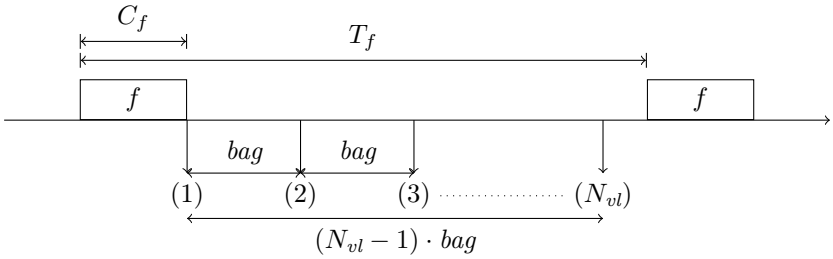


FIGURE 4.6 – Condition de stabilité d'un lisseur de trafic

Exemple 7 (Allocation des dépendances de l'étude de cas FMS)**.** *Une allocation bien formée des variables puits de l'étude de cas est donnée par la fonction $Alloc_D$ telle que :*

$$\begin{aligned}
Alloc_D(wpId_1, query_1) &= vl_1 \\
Alloc_D(wpId_1, query_2) &= vl_1 \\
Alloc_D(wpId_2, query_1) &= vl_2 \\
Alloc_D(wpId_2, query_2) &= vl_2 \\
Alloc_D(wpInfo_1, disp_1) &= vl_3 \\
Alloc_D(wpInfo_1, ETA_1) &= vl_3 \\
Alloc_D(query_1, answer_1) &= vl_4 \\
Alloc_D(wpInfo_2, disp_2) &= vl_5 \\
Alloc_D(wpInfo_2, ETA_2) &= vl_5 \\
Alloc_D(query_2, answer_2) &= vl_6 \\
Alloc_D(answer_1, wpInfo_1) &= vl_7 \\
Alloc_D(answer_2, wpInfo_2) &= vl_8
\end{aligned}$$

$$\begin{aligned}
Alloc_D(pres_1, speed_1) &= vl_9 \\
Alloc_D(pres_2, speed_2) &= vl_{10} \\
Alloc_D(speed_1, ETA_1) &= vl_{11} \\
Alloc_D(speed_2, ETA_2) &= vl_{12} \\
Alloc_D(\quad, pres_1) &= \\
Alloc_D(\quad, pres_2) &= \\
Alloc_D(disp_1, \quad) &= \\
Alloc_D(disp_2, \quad) &= \\
Alloc_D(\quad, req_1) &= \\
Alloc_D(\quad, req_2) &= \\
Alloc_D(req_1, wpId_1) &= \\
Alloc_D(req_2, wpId_2) &=
\end{aligned}$$

Les paramètres des liens virtuels alloués aux dépendances sont récapitulés dans le tableau 4.3.

4.4 Conclusion

Dans ce chapitre, nous avons présenté une formalisation d'un système IMA. Cette formalisation est composée d'un modèle d'architecture fonctionnelle, d'un modèle d'architecture matérielle et d'un modèle allocation. Nous avons également défini formellement les propriétés qui doivent être respectées pour que le système soit considéré comme étant bien formé.

Dans le chapitre suivant, nous décrivons l'ensemble des comportements possibles d'un système IMA bien formé à l'aide du *tagged signal model*.

VL	s_{min} (bits)	s_{max} (bits)	*paths*	*bag*
vl_1	600	600	$(M_1.1, S_1.1), (S_1.3, S_2.2), (S_2.1, M_3.1)$ $(M_1.1, S_1.1), (S_1.4, S_3.2), (S_3.1, M_4.1)$	32
vl_2	600	600	$(M_2.1, S_1.7), (S_1.4, S_3.2), (S_3.1, M_4.1)$ $(M_2.1, S_1.7), (S_1.3, S_2.2), (S_2.1, M_3.1)$	32
vl_3	1000	5000	$(M_3.1, S_2.1), (S_2.2, S_1.3), (S_1.1, M_1.1)$	8
vl_4	1000	1000	$(M_3.1, S_2.1), (S_2.2, S_1.3), (S_1.2, M_7.1)$	16
vl_5	1000	5000	$(M_4.1, S_3.1), (S_3.2, S_1.4), (S_1.7, M_2.1)$	8
vl_6	1000	1000	$(M_4.1, S_3.1), (S_3.2, S_1.4), (S_1.2, M_7.1)$	16
vl_7	4000	4000	$(M_7.1, S_1.2), (S_1.3, S_2.2), (S_2.1, M_3.1)$	64
vl_8	4000	4000	$(M_7.1, S_1.2), (S_1.4, S_3.2), (S_3.1, M_4.1)$	64
vl_9	512	512	$(R_1.1, S_4.3), (S_4.1, M_5.1)$	32
vl_{10}	512	512	$(R_2.1, S_5.3), (S_5.1, M_6.1)$	32
vl_{11}	700	700	$(M_5.1, S_4.1), (S_4.2, S_1.5), (S_1.3, S_2.2), (S_2.1, M_3.1)$ $(M_5.1, S_4.1), (S_4.2, S_1.5), (S_1.5, S_3.2), (S_3.1, M_4.1)$	32
vl_{12}	700	700	$(M_6.1, S_5.1), (S_5.2, S_1.6), (S_1.4, S_3.2), (S_3.1, M_4.1)$ $(M_6.1, S_5.1), (S_5.2, S_1.6), (S_1.3, S_2.2), (S_2.1, M_3.1)$	32

TABLE 4.3 – Caractéristiques des liens virtuels de l'étude de cas

Chapitre 5

Modèle comportemental d un système IMA

Dans ce chapitre, nous présentons un modèle décrivant l'ensemble des comportements possibles d'un système IMA. Nous nous intéressons à la fois au comportement temporel des composants du système et au lien de dépendance existant entre les différents événements du système. Le *tagged signal model* [75] est un formalisme très générique dont l'objectif originel est de capturer des propriétés essentielles de modèles de calcul pour pouvoir les comparer et les composer. Comme nous l'avons vu dans la présentation du *tagged signal model* (c.f. 2.2.2 page 30), nous pouvons détourner son utilisation pour capturer de manière compacte et relativement naturelle la dynamique de systèmes informatisés.

Dans ce contexte, la composition des processus est facilitée si tous les processus sont définis sur une même base de signaux. Nous commençons donc notre description en explicitant l'ensemble des signaux utilisés pour capturer le comportement temporel d'un système IMA. Ensuite, la modélisation des processus sera fondée sur cette base de signaux commune.

5.1 Dé nition des signaux utilisés

Un certain nombre de signaux sont nécessaires pour décrire le comportement d'un système. Plus précisément, il faut un signal par fonction pour modéliser son activation et pour chaque variable, il faut un signal pour chaque élément manipulant cette variable.

En effet, la modélisation des comportements d'une variable nécessite plusieurs signaux pour modéliser sa diffusion dans le système. Nous utilisons le terme *copie de variable* pour distinguer une variable (définie au niveau de l'architecture fonctionnelle) et ses utilisations dans les éléments du système. Par exemple, la variable *wpId* est produite par KU_1 (1 copie), puis traverse le lisseur de trafic de VL_1 (1 copie) puis le port de sortie du module M_1 (1 copie), puis est dupliquée dans le commutateur S_1 sur les ports $S_1.2$ et $S_1.3$ (2 copies). La figure 5.1 représente une partie des processus et signaux utilisés pour la variable *wpId*.

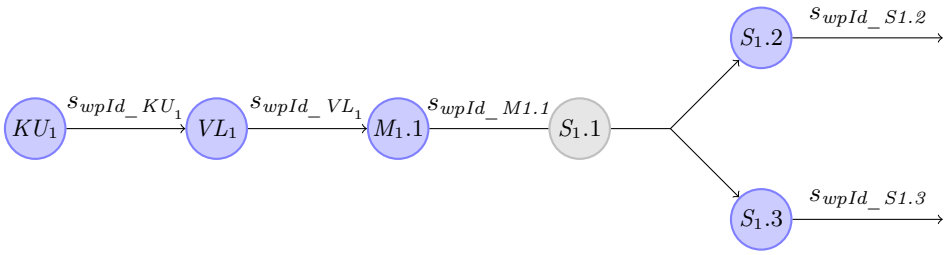


FIGURE 5.1 – Répartition des signaux utilisés pour la variable *wpId*

La place grisée $S_1.1$ représente le port d'entrée de VL_1 dans le commutateur S_1. La traversée d'un port dans ce sens retarde la communication d'un temps constant (la latence technologique du commutateur). Nous choisissons de ne pas modéliser ceci par un processus spécifique, cela sera pris en compte dans les

processus des ports de sortie. Le signal de la variable en sortie du port du module M_1 est donc directement « branché » sur les processus de $S_1.2$ et $S_1.3$.

Nous utilisons plusieurs conventions de nommage : le signal associé à l'activation de la fonction f est noté s_f et le signal d'une copie de la variable z en sortie d'un élément e est noté s_{z_e}.

5.1.1 Signaux pour les activations de fonction

Comme présenté en préambule de cette section, pour chaque fonction $f \quad Func$, nous utilisons un signal s_f dans la modélisation. Le nombre de signaux pour les fonctions n_F est donc le cardinal de $Func$:

$$n_F = card(Func) \tag{5.1}$$

5.1.2 Signaux pour les copies de variable

Pour la modélisation, il faut :

- pour chaque variable un signal en sortie de l'élément produisant la variable (capteur ou fonction),
- pour les variables destinées à l'environnement extérieur, un signal supplémentaire pour la sortie de l'actionneur,
- un signal pour chaque variable allouée à un capteur ou un actionneur relié à un RDC,
- pour chaque VL, un signal pour chaque variable passant par ce VL pour la sortie du lisseur de trafic, plus un signal pour chaque variable et pour chaque lien traversé par le VL pour les sorties des ports.

Le nombre de signaux n_V utilisés pour la modélisation des copies de variable du système est :

$$\begin{aligned} n_V =& card(Var) + card(sens_RDC) + card(act_RDC) \\ &+ \sum_{vl \quad VL} [card(varVL(vl)) \cdot (1 + n_{Links}(vl))] \end{aligned} \tag{5.2}$$

Avec :

- $sens_RDC = \quad s \quad Sensors \quad s.equipement \quad RDC$ l'ensemble des capteurs reliés à un RDC,
- $act_RDC = \quad a \quad Actuators \quad a.equipement \quad RDC$ l'ensemble des actionneurs reliés à un RDC,
- $\quad vl \quad VL$, $n_{Links}(vl)$ le nombre de liens couverts par le VL vl :

$$\begin{aligned} & vl \quad VL \text{ t.q. } vl = (s_{min}, s_{max}, paths, bag), \\ & \text{avec } vl.paths = \quad p_1, \ldots, p_n \quad \text{et} \quad i = 1..n \ , \ p_i = \quad l_1^i, \ldots, l_{m_i}^i \\ & n_{Links}(vl) = card(\bigcup_{i=1}^{n} \ l_1^i, \ldots, l_{m_i}^i \ ) \end{aligned} \tag{5.3}$$

5.1.3 Domaine de définition des signaux

Dans notre modèle, nous prendrons comme domaine des estampilles temporelles : $T = \mathbb{R}^+$. Pour le domaine des valeurs des événements, nous utilisons $V = (\mathbb{N} \quad)^N$, avec $N = n_F + n_V$ le nombre de signaux utilisés dans la modélisation (cf. formule 5.1 et 5.2). Ce domaine de valeurs nous permet de représenter les dépendances entre les événements comme présenté dans la section 2.2.2. Pour un événement i d'un signal s_j noté $s_j[i] = (t, v)$ nous encodons les dépendances dans le champ valeur v qui est un vecteur à N composantes $v = (d_1, \ldots, d_n)$ et il respecte : $\quad k = 1..N$ si $d_k = \quad$ cela signifie que $s_j[i]$ dépend du $d_k{}^{\text{ième}}$ événement du signal s_k.

Comme présenté dans la section 2.2.2, pour $s_j[i]$ le $i^{\text{ème}}$ événement d'un signal s_j, nous notons t_i^j son estampille et v_i^j sa valeur. Nous notons également $v_{i\ k}^j$ la valeur d_k. Si la valeur d'un événement est $(\quad , \ldots , \quad)$ alors il ne dépend d'aucun événement. Nous utilisons le symbole $\quad$ pour $(\quad , \ldots , \quad)$.

5.2 Dé nition des processus utilisés

Le comportement de chaque élément du système (module, fonction, lisseur de trafic, port, RDC, capteur ou actionneur) est décrit par un processus, c'est-à-dire un ensemble de contraintes entre signaux. Dans la suite, nous détaillons le comportement attendu de chacun de ces éléments et donnons leur modélisation dans le *tagged signal model*. Tous les processus sont définis sur la même base de signaux S^N.

5.2.1 Module

Une des hypothèses fondamentales des systèmes IMA est l'asynchronisme des modules. Les déphasages possibles entre les modules sont modélisés par un décalage (appelé *phase*) du démarrage de chaque module vis-à-vis de l'instant initial. Pour chaque module, en bornant la phase à l'hyper-période des fonctions hébergées par un module, on assure que l'ensemble des déphasages possibles entre modules est prise en compte dans le modèle.

Ainsi, pour chaque comportement d'un module $m \quad \mathit{Modules}$, il existe une phase $\phi_m \quad [0, \mathit{HP}(m)[$ à partir de laquelle l'ordonnancement des fonctions du module démarre. Tous les instants d'activation t_n^f d'une fonction f avec un offset O_f et une période T_f hébergée sur m sont donc de la forme :

$$t_n^f = \phi_m + O_f + n \cdot T_f \tag{5.4}$$

Plus précisément, pour un module $m \quad \mathit{Modules}$, le processus décrivant tous les signaux d'activation acceptables pour les fonctions hébergées par le module est défini par :

$$\begin{aligned} P_m = \ \mathbf{s} \quad S^N \quad & \phi_m \quad [0, \mathit{HP}(m)[, \\ & f \quad m.\mathit{host}, \\ & n \quad \mathbb{N},\ t_n^f = \phi_m + O_f + n \cdot T_f \end{aligned} \tag{5.5}$$

avec :

- $m.\mathit{host} = \quad f \quad \mathit{Func} \quad \mathit{Alloc}_F(f) = (m, T_f, O_f, C_f)$ l'ensemble des fonctions allouées sur le module m,
- O_f et T_f respectivement l'offset et la période de f,
- $\mathit{HP} = \mathit{ppcm}_{f \quad m.\mathit{host}}(T_f)$ l'hyper-période des fonctions de m,
- ϕ_m la phase du module qui permet de modéliser l'asynchronisme entre les modules.

Exemple 8. *Le module M_1 héberge les fonctions KU_1 et MFD_1 dont les allocations sont $\mathit{Alloc}_F(KU_1) = (M_1, 50, 0, 25)$ et $\mathit{Alloc}_F(MFD_1) = (M_1, 50, 25, 25)$. L'hyper-période de M_1 est égale à 50 ms et le processus de M_1 est :*

$$\begin{aligned} P_{M_1} = \ \mathbf{s} \quad S^N \quad & \phi_{M_1} \quad [0, 50[, \\ & n \quad \mathbb{N}, \\ & t_n^{KU_1} = \phi_{M_1} + n \cdot 50, \\ & t_n^{MFD_1} = \phi_{M_1} + 25 + n \cdot 50 \end{aligned} \tag{5.6}$$

Un exemple de comportement possible de M_1 est représenté sur la figure 5.2.

5.2.2 Fonction

Une fonction lit ses variables d'entrée lors de son activation. Passé cet instant, les mises à jour de ces variables d'entrée ne pourront pas être prises en compte avant la prochaine activation de la fonction. Pour une exécution donnée, l'écriture de toutes les variables en sortie se fait au même instant. Le comportement d'une fonction dépend de la nature (périodique ou sporadique) des variables de sortie qui lui sont associées. Pour faciliter la lecture de la modélisation, nous présentons trois variantes de fonction : la première est une fonction ne produisant que des variables périodiques, la deuxième est une fonction produisant des variables sporadiques qui ne dépendent que d'une unique variable en entrée, la troisième produit des variables sporadiques dépendant de plusieurs variables en entrée.

Rappelons que le traitement d'une occurrence de variable en entrée ne peut pas s'étaler sur plusieurs exécutions de la fonction. En outre, les occurrences en entrée sont toujours traitées pour produire les sorties : aucun mécanisme ne vient interrompre le fonctionnement normal d'une fonction.

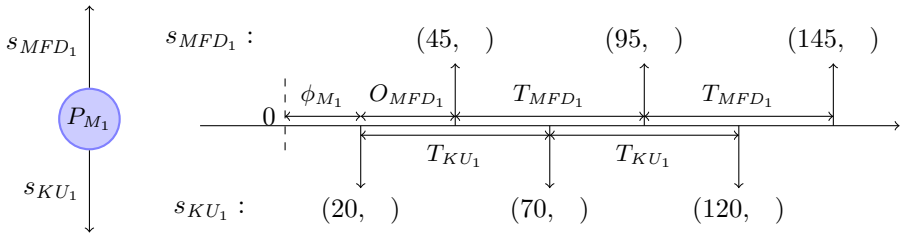


FIGURE 5.2 – Premiers événements d'un comportement de M_1 avec $\phi_{M_1} = 20$

Fonction avec n entrées par sortie périodique. Une variable périodique est produite à chaque période de la fonction, même si les variables dont elle dépend n'ont pas été rafraîchies. Ce sont systématiquement les dernières valeurs reçues de ces variables qui sont utilisées. Ceci correspond au comportement de la fonction MFD_1 décrit dans la section 2.2.2. Plus généralement, l'ensemble des comportements d'une fonction f $Func$ telle que v $f.Output$, $v.nat = periodic$ est défini avec les contraintes suivantes : après chaque exécution, la fonction produit toutes ses variables au même instant et avant son temps maximal de traitement C_f, ce qui peut s'exprimer :

$$n \quad \mathbb{N}, \quad t \quad [t_n^f, t_n^f + C_f], \quad v \quad f.Output, \; t_{n^{-f}}^{v} = t \tag{5.7}$$

avec t l'instant d'écriture des variables lors de la $n^{\text{ième}}$ exécution. En outre, comme les variables périodiques sont produites systématiquement à chaque période de la fonction, l'événement de la $n^{\text{ième}}$ occurrence d'une variable doit nécessairement dépendre du $n^{\text{ième}}$ événement d'activation de la fonction :

$$n \quad \mathbb{N}, \quad v \quad f.Output, \; v_{n^{-f}}^{v}{}_{f} = n \tag{5.8}$$

En outre, chaque événement pour une variable en sortie ne dépend que des derniers événements reçus de chacune des variables dont elle dépend :

$$\begin{aligned} & n \quad \mathbb{N}, \quad v \quad f.Output, \; u \quad f.Input \text{ tel que } (u,v) \quad Dep, \\ & v_{n^{-f}}^{v}{}_{u} = \max \; k \quad \mathbb{N} \quad t_k^u \leq t_n^f \end{aligned} \tag{5.9}$$

Nous pouvons alors synthétiser ces contraintes pour définir l'ensemble des comportements possibles de f avec le processus :

$$\begin{aligned} P_f = \; & \mathbf{s} \quad S^N \quad n \quad \mathbb{N}, \quad t \quad [t_n^f, t_n^f + C_f], \quad v \quad f.Output, \\ & t_{n^{-f}}^{v} = t \; , \\ & v_{n^{-f}}^{v}{}_{x} = \begin{cases} n \text{ si } x = f \\ \max \; k \quad \mathbb{N} \quad t_k^x \leq t_n^f \quad \text{si } x \quad f.Input \quad (x,v) \quad Dep \end{cases} \end{aligned} \tag{5.10}$$

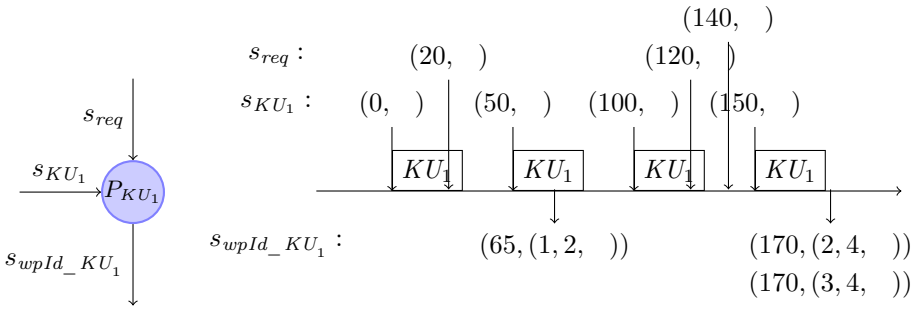
Si aucune copie d'une variable x n'a encore été reçue alors il n'y a pas de dépendance ($v_{n^{-f}}^{v}{}_{x} =$).

Fonction avec une entrée par sortie sporadique. Pour les variables sporadiques, nous traitons d'abord le cas des variables ayant une unique dépendance. Contrairement à une variable périodique, une variable sporadique est produite uniquement si la variable dont elle dépend a été mise à jour depuis la dernière activation de la fonction.

Exemple 9. *Nous décrivons sur la figure 5.3 un comportement possible de KU_1 réduit aux signaux s_{req}, s_{KU_1}, $s_{wpId_KU_1}$. Les signaux s_{KU_1} et s_{req} ne dépendent d'aucun autre signal. En revanche, $wpId_KU_1$ est produit par KU_1 et contient le traitement de la requête émise à 20ms. Il dépend donc des événements $s_{req}[1]$ et $s_{KU_1}[2]$.*

*La première requête du pilote est modélisée par $s_{req}[1]$. Elle arrive à 30ms et est traitée lors de la seconde exécution de KU_1. Nous supposons que le traitement prend ici $15ms \in [0, C_{KU_1}]$, avec C_{KU_1} le temps de traitement maximal de la fonction. A la fin du traitement, KU_1 produit une copie de la variable wpId (*waypoint identifier*) sur sa sortie. Ceci est modélisé par l'événement $s_{wp_KU_1}[1]$. La valeur de l'événement est $(1, 2, \quad)$ en respectant $s_{req}, s_{KU_1}, s_{wp_KU_1}$, cela signifie que $s_{wp_KU_1}[1]$ dépend des événements $s_{req}[1]$ et $s_{KU_1}[2]$ mais ne dépend d'aucun événement de $s_{wp_KU_1}$.*

Lors de la troisième exécution de KU_1, wpId (qui est une variable sporadique) n'est pas produite car aucune nouvelle requête n'a été reçue depuis la dernière activation de KU_1. Lors de la quatrième exécution, deux requêtes req ont été reçues, KU_1 produit alors deux copies de wpId.



(a) KU_1 (b) Premiers événements d'un comportement possible de KU_1

FIGURE 5.3 – Un comportement de la fonction KU_1

Soit $f \quad Func$ avec $u \quad f.Input$ et $v \quad f.Output$ telles que u soit la seule variable dont dépende la variable sporadique v. Comme chaque arrivée d'événements de la variable u provoque la production d'un événement en sortie sur v, le $n^{\text{ième}}$ événement de s_{v_f} dépend nécessairement du $n^{\text{ième}}$ événement de s_u.

$$n \quad \mathbb{N}, \; v_n^{v_f}{}_u = n \tag{5.11}$$

En outre, si le $n^{\text{ième}}$ événement de s_u provoquant $s_{v_f}[n]$ est traité lors de la $k^{\text{ième}}$ activation de f, alors $s_u[n]$ arrive après la $k-1^{\text{ième}}$ activation de f et avant la $k^{\text{ième}}$ activation et $s_{v_f}[n]$ dépend de cette activation :

$$n \quad \mathbb{N}, \quad k \quad \mathbb{N} \text{ tel que } t_n^u \quad [t_{k-1}^f, t_k^f[\text{ et } v_n^{v_f}{}_f = k \tag{5.12}$$

La fonction doit également produire ses sorties en un temps inférieur à C_f. Nous définissons alors le processus définissant l'ensemble des comportements possibles d'une telle fonction f :

$$\begin{aligned} P_f = \; & \mathbf{s} \quad S^N \quad n \quad \mathbb{N}, \; v \quad f.Output \text{ t.q. } (u, v) \quad Dep \quad v.nat = sporadic, \\ & k \quad \mathbb{N} \text{ tel que } t_n^u \quad [t_{k-1}^f, t_k^f[, \quad t \quad [t_k^f, t_k^f + C_f], \\ & t_n^{v_f} = t, \\ & v_n^{v_f}{}_f = k, \\ & v_n^{v_f}{}_u = n \end{aligned} \tag{5.13}$$

Si plusieurs mises à jour sont effectuées sur u entre deux exécutions de f, alors f traite toutes les copies reçues depuis sa dernière exécution : si il y a p événements de s_u pendant $]t_{k-1}^f, t_k^f]$ alors il y a p événements de s_{v_f} en sortie de la $k^{\text{ième}}$ exécution de f.

Fonction avec n entrées pour une sortie sporadique. Supposons maintenant que v dépende d'un ensemble de variables $u_1, ..., u_P$. Nous faisons le choix dans notre modélisation que n'importe

quel événement sur les variables $u_1, ..., u_P$ peut être déclencheur d'un événement sur v. Un exemple de comportement d'une telle fonction est représenté sur la figure 5.4 où v est une variable sporadique dépendant des variables u_1 et u_2. Les dépendances entre les occurrences de v et de u_1 et u_2 sont précisées avec une valeur $(i, (j, k))$ où i est l'indice d'une occurrence v, j l'indice de l'occurrence de u_1 utilisée dans le traitement et k celui de u_2.

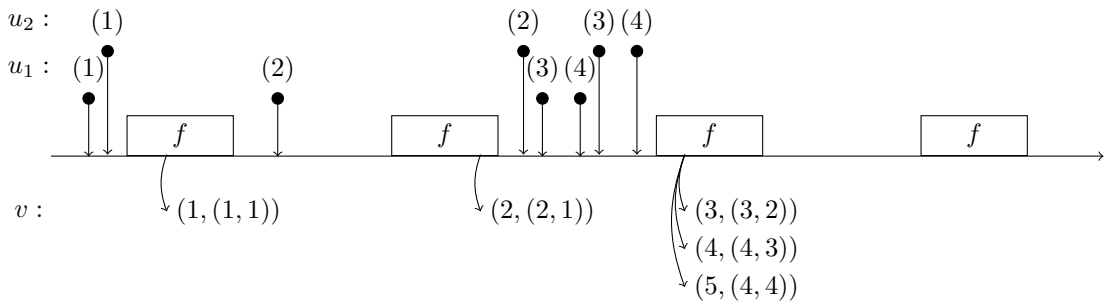


FIGURE 5.4 – Exemple d'une variable sporadique dépendant de deux variables

Nous formalisons le comportement de la fonction. Soit t_n^{v-f} l'estampille du $n^{\text{ième}}$ événement en sortie de f sur la variable v. Il existe une exécution de f produisant cet événement :

$$n \quad \mathbb{N}, \quad K \quad \mathbb{N} \text{ et } t \quad [0, C_f] \text{ tels que } t_n^{v-f} = t_K^f + t \tag{5.14}$$

Si l'exécution K est celle produisant l'événement n, alors elle doit respecter certaines contraintes. Notamment, il doit exister au moins un événement sur les variables en entrée entre le début de cette exécution et le début de la précédente. Pour formaliser cette contrainte, nous définissons la fonction $n_s(a, b)$ dénombrant les événements du signal s dont l'estampille appartient à l'intervalle $[a, b[$:

$$n_s(a, b) = card(n \quad \mathbb{N} \quad t_n^s \quad [a, b[) \tag{5.15}$$

Par exemple, sur le comportement représenté sur la figure 5.4, le nombre d'événements sur u_1 entre les débuts des deuxième et troisième exécutions de f est $n_{u_1}(t_2^f, t_3^f) = 2$.

Pour qu'il y ait une occurrence de v produite lors de l'exécution K, il faut qu'il y ait au moins un événement en entrée depuis la dernière exécution :

$$i \quad 1..P \text{ tel que } n_{u_i}(t_{K-1}^f, t_K^f) \geq 1 \tag{5.16}$$

A noter que cette condition peut également s'écrire de la manière suivante :

$$\sum_{i=1}^{P} n_{u_i}(t_{K-1}^f, t_K^f) \geq 1 \tag{5.17}$$

Le nombre d'événements en sortie produits lors d'une exécution quelconque k correspond au plus grand nombre d'événements arrivés sur une des entrées. Par exemple, si f a deux entrées, avec un événement sur l'une et trois événements sur l'autre, alors il y aura trois événements en sortie. Le nombre d'événements en sortie d'une exécution k est donc :

$$\max_{i=1..P} \; n_{u_i}(t_{k-1}^f, t_k^f) \tag{5.18}$$

Pour l'exemple représenté sur la figure 5.4, on a $\max_{i=1..2} \; n_{u_i}(t_1^f, t_2^f) \;\; = \max \; 1; 0 \;\; = 1$ et donc une unique occurrence de v est produite lors de la deuxième exécution de f. On a également $\max_{i=1..2} \; n_{u_i}(t_2^f, t_3^f) \;\; =$

$\max\ 2;3\ = 3$ donc trois occurrences de v sont produites lors de la troisième exécution de f et finalement $\max_{i=1..2}\ n_{u_i}(t_3^f, t_5^f)\ = 0$ et donc aucune occurrence de v n'est produite lors de la quatrième exécution de f.

On peut alors lier l'indice de l'exécution K avec la production de la $n^{\text{ième}}$ occurrence de v de la façon suivante : l'exécution K est telle que l'ensemble des exécutions précédentes a produit strictement moins de n événements en sortie, et après l'exécution K il y a au moins n événements en sortie.

$$\sum_{k=1}^{K-1} \max_{i=1..P}\ n_{u_i}(t_{k-1}^f, t_k^f)\ < n \leq \sum_{k=1}^{K} \max_{i=1..P}\ n_{u_i}(t_{k-1}^f, t_k^f) \tag{5.19}$$

Le membre de gauche de cette inéquation correspond à l'indice de la dernière occurrence de v produite avant l'exécution K, et le membre de droite à l'indice de la dernière occurrence produite lors de l'exécution K. Dans la suite, nous utilisons l'indice de la dernière occurrence produite avec l'exécution K et nous la notons $L(K)$.

Nous déterminons les dépendances entre l'occurrence produite lors de l'exécution K et les occurrences en entrée. La difficulté ici est de définir la façon dont les occurrences en entrée sont associées pour produire une occurrence en sortie. Tout d'abord, considérons la première occurrence produite lors de l'exécution K et précisons l'indice de l'occurrence de la variable u_i dont elle dépend. Deux cas de figure sont possibles : soit aucun nouvel événement sur u_i n'est arrivé depuis la dernière exécution de f et alors la sortie dépend du dernier événement, soit il existe au moins un nouvel événement et la sortie dépend du premier événement reçu depuis la dernière exécution. On a alors :

$$v_{L(K)+1\ u_i}^{v_f} = \sum_{k=1}^{K-1} n_{u_i}(t_{k-1}^f, t_k^f) + \min\ 1; n_{u_i}(t_{K-1}^f, t_K^f) \tag{5.20}$$

Si aucun événement sur u_i n'a été reçu depuis la dernière exécution, alors $n_{u_i}(t_{K-1}^f, t_K^f) = 0$ et $\min\ 1; n_{u_i}(t_{K-1}^f, t_K^f)\ = 0$. L'indice de l'événement dont dépend la première sortie $v_{L(K)+1\ u_i}^{v_f}$ est alors égale à l'indice du dernier événement reçu.

Plus généralement, pour la $p^{\text{ième}}$ sortie de l'exécution K, l'indice de la dépendance avec u_i vaut :

$$v_{L(K)+p\ u_i}^{v_f} = \sum_{k=1}^{K-1} n_{u_i}(t_{k-1}^f, t_k^f) + \min\ p; n_{u_i}(t_{K-1}^f, t_K^f) \tag{5.21}$$

La $n^{\text{ième}}$ occurrence de v correspond au cas $p = n - L(K)$.

Le processus d'une telle fonction est alors :

$$\begin{aligned}
P_f = \ \mathbf{s} \quad S^N \quad & n \quad \mathbb{N}, v \quad f.Output, v.nat = sporadic, \ \ i = 1..P \text{ t.q. } (u_i, v) \quad Dep \\
& K \quad \mathbb{N}, \ t \quad [0, C_f], \\
& t_n^{v_f} = t_K^f + t, \\
& \text{avec } \sum_{k=1}^{K-1} \max_{i=1..P}\ n_{u_i}(t_{k-1}^f, t_k^f)\ < n \leq \sum_{k=1}^{K} \max_{i=1..P}\ n_{u_i}(t_{k-1}^f, t_k^f)\ , \\
& v_{n\ x}^{v_f} = \begin{cases} K \text{ si } x = f \\ \sum_{k=1}^{K-1} n_{u_i}(t_{k-1}^f, t_k^f) + \min\ n - L(K); n_{u_i}(t_{K-1}^f, t_K^f) & \text{si } x = u_i \\ \text{sinon} \end{cases}
\end{aligned} \tag{5.22}$$

Remarque : nous ne nous intéressons pas ici à déterminer le nombre maximal d'événements traitables par une fonction à chaque période. Il s'agit d'un problème de dimensionnement de la plateforme qui dépasse le cadre de cette thèse. Nous supposons qu'il s'agit d'un paramètre connu et respecté : une fonction ne recevra jamais plus de copies de variables qu'elle ne peut en traiter. Ce paramètre intervient également dans le dimensionnement des lisseurs de trafic.

5.2.3 Lisseur de trafic

Nous supposons qu'il y a une unique occurrence de variable dans chaque paquet transmis au travers d'un lien virtuel. Le rôle du lisseur de trafic est alors d'assurer qu'il existe une distance minimale entre chaque occurrence des variables envoyées sur le réseau. Considérons le lisseur de trafic de VL transmettant les variables $v_1, \ldots, v_l$ produites par la fonction f. Les signaux directement affectés par le processus du lisseur de trafic sont représentés sur la figure 5.5. A noter qu'un lisseur de trafic peut également transmettre des variables produites par un RDC. Cela n'a pas d'impact sur la modélisation du lisseur de trafic.

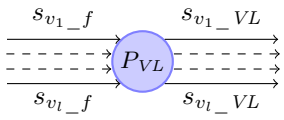


FIGURE 5.5 – Signaux en relation dans le processus P_{VL}

Le lisseur de trafic régule les occurrences de variable dans leur ordre d'arrivée en suivant une politique de service « premier arrivé, premier servi » (FIFO). Mais plusieurs occurrences peuvent arriver simultanément. Dans ce cas, il faut considérer tous les ordres de traitement possibles. Sur la figure 5.6, deux ordonnancements possibles d'une même arrivée sont représentés. Les flèches en direction de la ligne des temps représentent des arrivées dans le lisseur de trafic. Ici, les arrivées sont numérotées pour permettre d'identifier facilement les sorties associées. Le lisseur de trafic traite les arrivées indépendamment de leur origine. Ainsi sur la figure nous ne distinguons pas les variables correspondant aux différentes arrivées. Sur ces exemples, on peut observer la régulation du lisseur de trafic : il y a une sortie au plus tous les *bag* unités de temps. Les sorties se font dans l'ordre d'arrivée, à l'exception des arrivées (4) et (5) qui sont simultanées. Pour ces arrivées, deux ordres de sortie sont possibles : (4)-(5) et (5)-(4).

A noter que nous modélisons un lisseur de trafic en toute généralité : nous ne prenons pas en compte l'hypothèse qu'une fonction produit toutes ses sorties simultanément. Nous prendrons en compte cette hypothèse (qui simplifiera la modélisation) lorsque nous construirons une abstraction du système (c.f. section 7.1.3 page 111).

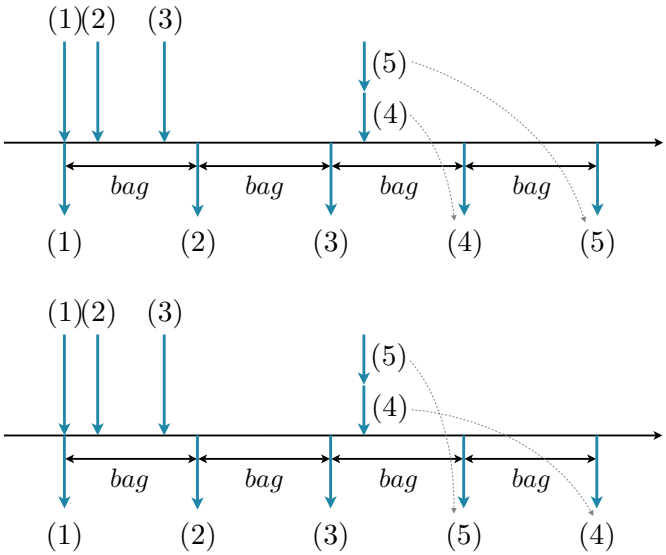


FIGURE 5.6 – Deux sorties possibles d'un lisseur de trafic pour une même arrivée

Pour formaliser le comportement d'un lisseur de trafic, considérons la $n^{\text{ième}}$ arrivée de la variable v_i dont l'estampille est $t_n^{v_i-f}$. Cette occurrence peut être retardée par deux facteurs :

- des occurrences arrivées avant elle non encore expédiées,
- des occurrences arrivées en même temps qu'elle et prises en compte avant elle.

Par exemple, sur le chronogramme en haut de la figure 5.6, l'arrivée (4) doit attendre que le *bag* de l'émission précédente (3) soit écoulé avant d'être expédiée. L'arrivée (5) doit attendre un *bag* supplémentaire car (4) est prise en compte avant elle.

Le temps d'attente dû aux occurrences arrivées précédemment et n'ayant pas encore été expédiées est appelé la *charge de travail*. La charge de travail à un instant t est notée $W(t)$ (W pour *workload*). Nous notons le temps d'attente dû aux occurrences arrivant simultanément $I(v, n)$ (I pour *interférence*). Ce temps d'attente est défini pour une variable v et une occurrence n. Il ne dépend pas d'un temps quelconque t car il n'a de sens qu'à l'instant d'arrivée de l'occurrence.

Ainsi, on peut définir l'instant de sortie de la $n^{\text{ième}}$ occurrence de la variable v_i de la façon suivante :

$$t_n^{v_i-VL} = t_n^{v_i-f} + W(t_n^{v_i-f}) + I(v_i, n) \tag{5.23}$$

Nous représentons sur la figure 5.7 la charge de travail et l'interférence subie par l'arrivée (4). La date de cette arrivée est notée $t_4^{v_i-f}$, la date de la sortie associée $t_4^{v_i-VL}$, la charge de travail lors de l'arrivée est $W(t_4^{v_i-f})$ et l'interférence subie est $I(v_i, 4)$.

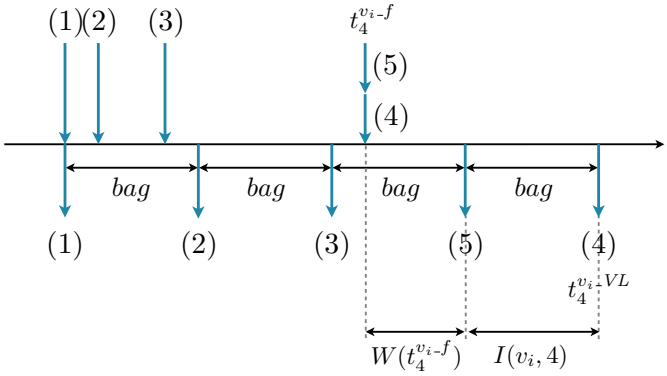


FIGURE 5.7 – Charge de travail et interférence pour l'arrivée (4)

Pour un ensemble d'arrivées donné, nous commençons par déterminer la valeur de la charge de travail $W(t_n^{v_i-f})$. Ensuite, nous déterminerons l'ensemble des valeurs possibles de l'interférence $I(v, n)$.

Charge de travail $W(t)$. La figure 5.8 représente l'évolution de la charge de travail du lisseur de trafic pour les arrivées représentées sur la figure 5.6. Lors de l'arrivée (1) à l'instant 1, la charge de travail du lisseur de trafic est nulle. L'arrivée est donc expédiée instantanément et la charge de travail augmente d'un *bag* (sur cet exemple, *bag* = 4). Le *bag* correspond au temps d'attente minimal entre deux envois successifs, la charge de travail décroît donc linéairement (pour x unités de temps écoulé, le charge de travail diminue de x) jusqu'à la prochaine arrivée. L'arrivée (2) a lieu à l'instant 2, la charge de travail a donc diminué de 1 et est égale à 3. Cette valeur correspond au retard que subira cette arrivée. La charge de travail est augmentée d'un *bag*. Ainsi lors de l'arrivée (3) à l'instant 4, la charge de travail vaut 5 et donc (3) sortira à 9. A l'instant 10, les arrivées (4) et (5) ont lieu et ainsi la charge augmente de 2 *bags*. Les instants de sortie de ces arrivées dépendront de leur ordre de traitement par la lisseur de trafic, ce qui sera définit par la notion d'interférence subie par une arrivée.

Pour déterminer la valeur de la charge de travail à un instant t ($W(t)$), il faut connaître les événements survenus avant t. Pour éviter d'avoir à prendre en compte l'ensemble des événements depuis l'initialisation du système, la notion de *période d'activité* est classiquement utilisée. On retrouve cette notion sous différentes formes dans les travaux de Lehoczky [77], Tindell [105] ou encore Migge [86, 87]. Une période d'activité est un intervalle de temps pendant lequel la charge de travail n'est jamais nulle. La figure 5.8 représente une période d'activité débutant après l'arrivée de (1) et finissant après l'envoi de (5) lorsque la charge de travail redevient nulle. L'idée derrière la notion de période d'activité est que tout événement précédant une période d'activité ne peut avoir d'influence sur celle-ci. On limite ainsi le nombre d'événements à considérer.

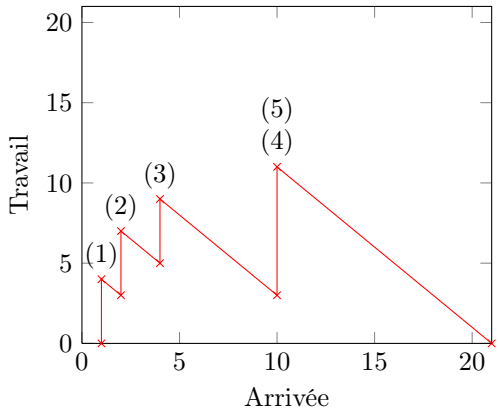


FIGURE 5.8 – Fonction de travail du lisseur de trafic

On s'intéresse donc au début de la période d'activité contenant l'instant t. Dans notre cas, le début de la période coïncide nécessairement avec l'arrivée d'une occurrence de variable ; notons $t_{n'}^{v_j-f}$ l'estampille de cet événement. La charge de travail à l'instant t est égale à l'ensemble du travail reçu entre $t_{n'}^{v_j-f}$ et t moins tout le travail effectué entre $t_{n'}^{v_j-f}$ et t.

Chaque événement en entrée du lisseur de trafic arrivant dans l'intervalle $[t_{n'}^{v_j-f}, t[$ apporte une charge de travail d'un *bag*. Nous notons $\mathcal{W}_{v_k_f}(t_{n'}^{v_j-f}, t)$ la charge de travail apportée sur l'intervalle $[t_{n'}^{v_j-f}, t[$ par le signal $s_{v_k_f}$:

$$\mathcal{W}_{v_k_f}(t_{n'}^{v_j-f}, t) = card(n \quad \mathbb{N} \quad t_{n'}^{v_j-f} \leq t_n^s < t) \cdot bag \tag{5.24}$$

Ainsi la charge de travail apportée par l'ensemble des occurrences des variables $v_1, \ldots, v_l$ sur cet intervalle est :

$$\sum_{k=1}^{l} \mathcal{W}_{v_k_f}(t_{n'}^{v_j-f}, t) \tag{5.25}$$

Comme il n'existe pas de période d'inactivité sur l'intervalle $[t_{n'}^{v_j-f}, t[$ (par définition), le travail effectué sur l'intervalle est égal à la longueur de l'intervalle, soit :

$$t - t_{n'}^{v_j-f} \tag{5.26}$$

On en déduit une expression de la charge de travail à l'instant t :

$$W(t) = \sum_{k=1}^{l} \mathcal{W}_{v_k_f}(t_{n'}^{v_j-f}, t) - (t - t_{n'}^{v_j-f}) \tag{5.27}$$

avec $t_{n'}^{v_j-f}$ l'instant du début de la période d'activité contenant t.

Pour identifier l'événement à l'origine d'une période d'activité, on fait l'observation suivante : seules les occurrences à l'origine de périodes d'activité sont transférées instantanément car aucun événement ne les retarde. Ainsi, l'événement à l'origine de la période d'activité contenant t doit respecter les contraintes suivantes :

$$\begin{aligned} t_{n'}^{v_j-f} &= \max_{i=1..l,\ k \ \mathbb{N}} t_k^{v_i-f} \ t_k^{v_i-f} \leq t \\ t_{n'}^{v_j-f} &= t_{n'}^{v_j-VL} \end{aligned} \tag{5.28}$$

La première contrainte de (5.28) stipule que l'origine de la période d'activité contenant t débute avant t et qu'elle est le dernier démarrage de période d'activité avant t.

A noter qu'il existe des périodes de temps creux (ou d'inactivité) car la conception du système nous garantit que le lisseur de trafic est bien formé (c.f. allocation des dépendances, sous section 4.3.3 page 71).

Interférence $I(v,n)$. L'interférence subie par l'occurrence n de la variable v_i dépend du nombre d'occurrences arrivant simultanément et prises en compte avant elle. Dans un premier temps, nous déterminons le nombre d'occurrences total arrivant simultanément. Pour une variable v_k, le nombre d'occurrences arrivant à un instant t est :

$$card(n \quad \mathbb{N} \quad t_n^{v_k-f} = t) \tag{5.29}$$

Nous définissons alors la fonction $n_I(t)$ qui dénombre les occurrences des variables $v_1, \ldots, v_l$ arrivant à l'instant t.

$$n_I(t) = \sum_{k=1}^{l} card(n \quad \mathbb{N} \quad t_n^{v_k-f} = t) \tag{5.30}$$

Ainsi le nombre d'occurrences arrivant à l'instant $t_n^{v_i-f}$ pouvant interférer avec l'occurrence n de v_i_f est compris entre 0 et $n_I(t_n^{v_i-f}) - 1$ (une occurrence ne peut pas interférer avec elle-même). On peut alors exprimer un comportement par :

$$c_n^i = 0..n_I(t_n^{v_i-f}) - 1 \text{ t.q. } I(v_i,n) = bag \cdot c_n^i \tag{5.31}$$

En outre, toutes les occurrences arrivant à l'instant $t_n^{v_i-f}$ doivent avoir une valeur d'interférence différente (sinon, certaines sortiraient en même temps du lisseur de trafic). Ceci se traduit par la contrainte suivante :

$$(j, j\ ) = 1..l\ , \quad (p, p\ ) \quad \mathbb{N}^2,\ ((p = p \quad j = j\ ) \quad t_p^{v_j-f} = t_{p'}^{v_{j'}-f}) \ = \quad I(v_j,p) = I(v_{j'},p\ ) \tag{5.32}$$

Instant de sortie d une occurrence. L'instant de sortie est alors la somme de l'instant d'arrivée, de la charge de travail à cet instant et de l'interférence :

$$t_n^{v_i-VL} = t_n^{v_i-f} + W(t_n^{v_i-f}) + I(v_i,n) \tag{5.33}$$

Processus du lisseur de tra c. Le processus d'un lisseur de trafic VL régulant les variables $v_1, \ldots, v_l$ produites par une fonction f est :

$$\begin{aligned} P_{VL} = \ \mathbf{s} \quad S^N \quad & i = 1..l, \\ & n \quad \mathbb{N}, \\ & t_n^{v_i-VL} = t_n^{v_i-f} + W(t_n^{v_i-f}) + I(v_i,n), \\ & v_n^{v_i-VL}{}_{v_i_f} = n \end{aligned} \tag{5.34}$$

Du fait de l'utilisation de $W(t_n^{v_i-f})$ et de $I(v_i,n)$, la définition de ce processus est compliquée. Mais en pratique, dans l'analyse globale et avec les hypothèses faites sur les fonctions, son expression pourra être simplifiée. Cette simplification est présentée dans la section 7.1.3.

5.2.4 Port

Comme présenté en introduction de la définition des signaux utilisés (section 5.1) nous ne modélisons les ports que dans le sens sortant. Deux cas de figure sont possibles :

- le port est en sortie d'un module et il agrège des occurrences de variables issues des lisseurs de trafic du module,
- le port fait partie d'un commutateur et il agrège des occurrences de variables issues de ports d'autres commutateurs, de ports de modules ou de RDCs.

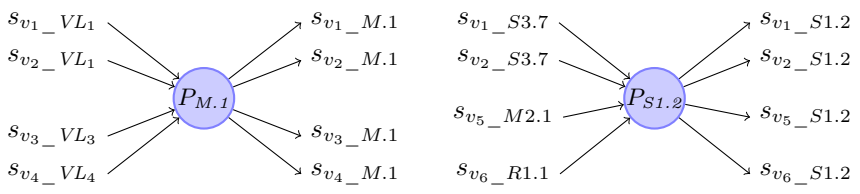


FIGURE 5.9 – Signaux en relation dans le processus d'un port

Sur la figure 5.9 ces deux cas de figure sont représentés. A gauche, le processus du port $M.1$ du module M a en entrée les signaux des variables en sortie des lisseurs de trafic des VL VL_1, VL_3 et VL_4. A droite, il s'agit du processus du port $S1.2$ du commutateur $S1$. Ce port a en entrée les variables v_1 et v_2 en sortie du port $S3.7$ du commutateur $S3$, la variable v_5 en sortie du port $M2.1$ du module $M2$ et la variable v_6 en sortie du port $R1.1$ du RDC R1.

La différence entre ces deux cas de figure est qu'il existe une latence lors du routage d'une trame dans un commutateur, alors qu'une occurrence en sortie d'un lisseur de trafic est instanément copiée dans le port de sortie d'un module. Nous utilisons une même modélisation prenant en compte une latence technologique dans les deux cas. Dans le cas d'un port d'un module, la latence devra être considérée nulle.

Pour homogénéiser la présentation de la modélisation, nous renommons les entrées $I_1, \ldots, I_K$ et les sorties $O_1, \ldots, O_K$ avec K le nombre de signaux en entrée du processus. Par exemple, $s_{v_1_VL_1}$ devient s_{I_1} et $s_{v_1_M.1}$ devient s_{O_1} dans la figure 5.9 à gauche. Nous utilisons la notation $var(I_k)$ (respectivement $var(O_k)$) pour désigner la variable de l'entrée I_k (respectivement la sortie O_k). Ce renommage est tel que $var(I_k) = var(O_k)$.

La modélisation d'un port est similaire à la modélisation d'un lisseur de trafic : les occurrences de variables sont traitées dans leur ordre d'arrivée et pour les occurrences arrivant simultanément, tous les ordres de traitement doivent être considérés. La modélisation repose sur les notions de charge de travail et d'interférence. Il y a cependant trois différences :

- toutes les arrivées sont retardées d'une durée *lag* avant d'être prises en compte (une occurrence arrivant à un instant t_n^i ne sera prise en compte qu'à l'instant $t_n^i + lag$),
- la charge de travail d'une occurrence dépend de la taille de la variable (pour une variable v de taille $v.size$ et un port avec un débit d, la charge de travail d'une occurrence correspond à son temps de transmission, soit $\frac{v.size}{d}$),
- une occurrence est considérée transmise quand sa charge de travail est consommée.

La latence technologique *lag* retarde chaque arrivée avec un même délai *lag*. Ce délai n'a donc pas d'impact sur l'ordre de la prise en compte des occurrences par le port. Il est par conséquent équivalent d'appliquer ce retard en entrée ou en sortie du port, ce qui est illustré sur la figure 5.10. Nous faisons le choix d'appliquer ce retard en sortie, ce qui permet de ne pas sur-charger les fonctions d'interférence et de charge de travail.

Nous précisons désormais les différences par rapport au modèle d'un lisseur de trafic.

Charge de travail $W(t)$. La charge de travail apportée par la $n^{\text{ième}}$ occurrence d'une entrée I_k correspond au temps de transmission d'une occurrence de la variable $var(I_k)$, soit :

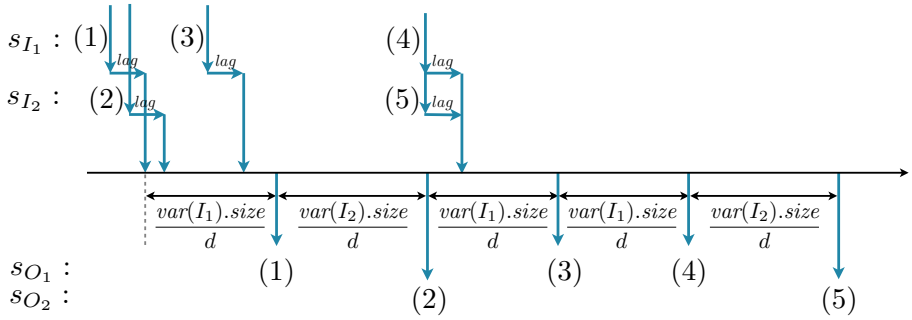
$$\frac{var(I_k).size}{d} \tag{5.35}$$

Cette charge de travail est identique quelque soit l'occurrence de I_k car les variables sont supposées de taille constante (c.f. section 4.1 page 61). Ainsi, la charge de travail apportée par l'entrée I_k pendant un intervalle $[a, b[$ est :

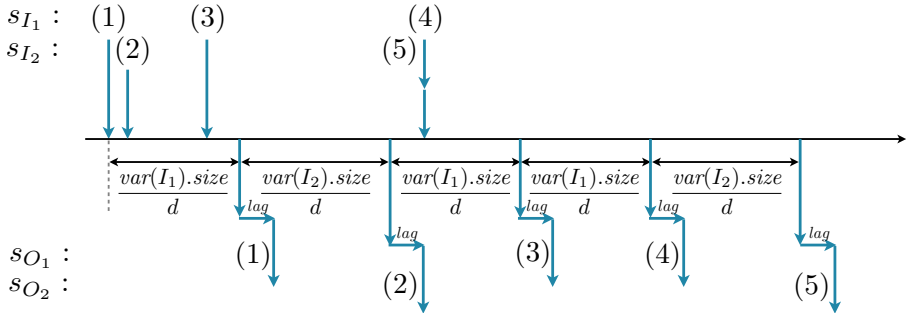
$$\mathcal{W}_{I_k}(a, b) = card(n \quad \mathbb{N} \quad a \leq t_n^{I_k} < b) \cdot \frac{var(I_k).size}{d} \tag{5.36}$$

Comme pour le lisseur de trafic, on peut en déduire une expression de la charge de travail à l'instant t :

$$W(t) = \sum_{k=1}^{K} \mathcal{W}_{I_k}(t_{n'}^{I_j}, t) - (t - t_{n'}^{I_j}) \tag{5.37}$$



(a) Prise en compte avant le traitement



(b) Prise en compte après le traitement

FIGURE 5.10 – Prise en compte de la latence technologique d'un port

avec $t_{n'}^{I_j}$ l'instant initial de la période d'activité contenant t. L'occurrence correspondant au début de cette période est telle que :

$$t_{n'}^{I_j} = \max_{i=1..K,\ k\ \ \mathbb{N}} t_k^{I_i}\ t_k^{I_i} \leq t \tag{5.38}$$

En outre, cette occurrence n'est retardée que par la latence technologique et son propre temps de transmission :

$$t_{n'}^{O_j} = t_{n'}^{I_j} + \frac{v_j.size}{d} + lag \tag{5.39}$$

Interférence $I(I_i, n)$. Comme pour le lisseur de trafic, l'interférence subie par la $n^{\text{ième}}$ occurrence sur l'entrée I_i est définie à partir du nombre d'occurrences arrivant en même temps qu'elle. En revanche, comme le temps de transmission peut être différent pour chaque variable, il faut différencier le nombre d'arrivée à l'instant $t_n^{I_i}$ pour chaque variable. On définit alors la fonction $n_I(t, k)$ retournant le nombre d'occurrences de l'entrée I_k arrivant à l'instant t :

$$n_I(t, k) = card(n \quad \mathbb{N} \quad t_n^{I_k} = t) \tag{5.40}$$

Nous exprimons l'interférence subie par la $n^{\text{ième}}$ occurrence de l'entrée I_i. Nous notons $c_n^{I_i}(I_k)$ le nombre d'occurrences de I_k étant traitées dans le port avant la $n^{\text{ième}}$ occurrence de I_i. Cette occurrence subit au minimum l'interférence d'une occurrence (elle-même et on a ainsi $c_n^{I_i}(I_i) \geq 1$) et au maximum l'interférence de toutes les occurrences arrivant au même instant qu'elle (pour toute entrée I_k, on a alors $c_n^{I_i}(I_k) \leq n_I(t_n^{I_i}, k)$). On peut alors exprimer un comportement comme la somme des interférences de chaque entrée :

$$\begin{aligned} & k = 1..K, k = i, \quad c_n^{I_i}(I_k) = 0..n_I(t_n^{I_i}, k), \quad c_n^{I_i}(I_i) = 1..n_I(t_n^{I_i}, i), \\ & \text{t.q. } I(I_i, n) = \sum_{k=1}^{K} \frac{v_k.size}{d} \cdot c_n^{I_i}(I_k) \end{aligned} \tag{5.41}$$

Instant de sortie L'instant de sortie est alors la somme de l'instant d'arrivée, de la charge de travail à cet instant, de l'interférence et de la latence technologique. Nous rappelons qu'il existe $\Pi_{k=1}^{K}(n_I(t_n^{I_i}, k) + 1) - 1$ valeurs possibles d'interférence.

$$t_n^{O_i} = t_n^{I_i} + W(t_n^{I_i}) + I(I_i, n) + lag \tag{5.42}$$

Dé nition du processus Le processus d'un port p s'exprime donc par :

$$\begin{aligned} P_p = \ & \mathbf{s} \quad S^N \quad i = 1..K, \quad n \quad \mathbb{N}, \\ & t_n^{O_i} = t_n^{I_i} + W(t_n^{I_i}) + I(I_i, n) + lag \\ & v_n^{O_i}{}_{I_i} = n \end{aligned} \tag{5.43}$$

5.2.5 Capteur

Un capteur peut avoir deux comportements différents suivant qu'il produit une variable périodique ou sporadique. Nous modélisons tout d'abord le cas d'une variable périodique.

Capteur produisant une variable périodique. Soit une variable *périodique* $v \quad Var$, allouée à un capteur $s \quad Sensors$ c'est-à-dire $Alloc_S(v) = s$. Le capteur est caractérisé par une période de production de la variable T et par un intervalle $[a, b]$ représentant le temps de traversée du bus de terrain reliant le capteur à son équipement (module ou RDC).

Tout comme un module, un capteur est un équipement asynchrone, il est donc sujet à un déphasage vis-à-vis des autres équipements. Nous modélisons ce déphasage par un décalage (aussi nommé *phase*) par rapport à l'instant initial. A la suite de ce déphasage, le capteur produit périodiquement sa variable qui est acheminée à son équipement en un temps δ compris dans l'intervalle $[a, b]$. Les comportements possibles d'un tel capteur s peuvent être modélisés par le processus :

$$\begin{aligned} P_s = \ & \mathbf{s} \quad S^N \quad \phi_s \quad [0, T[, \quad n \quad \mathbb{N}, \quad \delta \quad [a, b], \\ & t_n^{v-s} = \phi_s + n \cdot T + \delta \end{aligned} \tag{5.44}$$

avec :
- $\delta \quad [a, b]$ le délai imposé à une occurrence de la variable par le bus de terrain,
- $\phi_s \quad [0, T[$ la phase du capteur.

Capteur produisant une variable sporadique. Soit une variable *sporadique* $v \quad Var$, allouée à un capteur $s \quad Sensors$ tel que $Alloc_S(v) = s$. Notons T la distance minimale séparant deux productions consécutives de la variable et $[a, b]$ l'intervalle représentant le temps de traversée du bus de terrain reliant le capteur à son équipement (module ou RDC).

La modélisation des comportements du capteur se fait sur le même principe que pour le capteur périodique. Dans le cas sporadique, il existe un temps minimal entre deux productions, mais il n'y a pas de contrainte de périodicité. Autrement dit, il n'y a pas d'obligation de production de variable. Pour la première production, nous pouvons établir une contrainte vis-à-vis de la phase du capteur (on considère que la variable est produite lorsqu'elle est visible en sortie du bus de terrain) :

$$\phi_s \quad [0, T[\text{ tel que } t_0^{v-s} \geq \phi_s + a \tag{5.45}$$

Cette contrainte donne la date au plus tôt de la première production. Ensuite, les autres estampilles s'expriment vis-à-vis des estampilles précédentes. En particulier, il existe une distance minimale entre deux productions successives comme illustré sur la figure 5.11. Deux productions successives sont représentées sur la ligne en bas de la figure. La distance minimale entre deux productions au niveau du capteur est par définition T. La distance minimale en sortie du bus arrive alors lorsque la première occurrence traverse le bus en un temps maximal (b) et l'occurrence suivante en un temps minimal (a). La distance minimale entre deux occurrences en sortie du l'ensemble capteur/bus est alors de $T - (b - a)$.

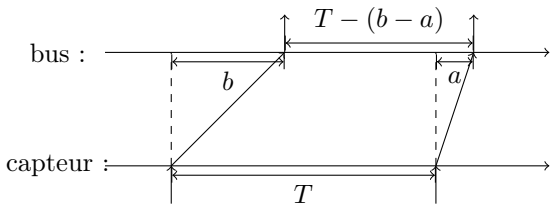


FIGURE 5.11 – Distance minimale entre deux occurrences successives en sortie d'un capteur

La distance minimale entre deux occurrences ne suffit pas pour définir la distance minimale entre k occurrences successives. Nous représentons le cas $k = 4$ sur la figure 5.12.

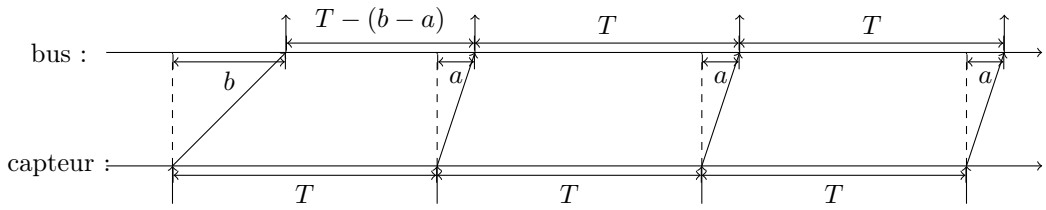


FIGURE 5.12 – Distance minimale entre k occurrences successives en sortie d'un capteur

La distance minimale entre une occurrence n et l'occurrence $n-k$ est ainsi la distance minimale entre k productions au niveau du capteur $(k-1) \cdot T$ moins la gigue maximale induite par le bus $(b-a)$. On en déduit donc la contrainte suivante pour les estampilles d'une occurrence n et d'une occurrence $n-k$:

$$n \quad \mathbb{N}^*,\ t_n^{v_s} \geq t_{n-k}^{v_s} + (k-1) \cdot T - (b-a) \tag{5.46}$$

Le processus associé au capteur sporadique s est alors :

$$\begin{array}{llll} P_s = & \mathbf{s} & S^N & \phi_s \quad [0, T[,\ t_0^{v_s} \geq \phi_s + a \\ & & & (n,k) \quad \mathbb{N}^2,\ t_n^{v_s} \geq t_{n-k}^{v_s} + (k-1) \cdot T - (b-a) \end{array} \tag{5.47}$$

5.2.6 Actionneur

Soit un actionneur $A \quad Actuators$ auquel est allouée une variable $v \quad Var$, c'est-à-dire $Alloc_P(v) = A$. Une occurrence de la variable v envoyée à un actionneur traverse tout d'abord un bus de terrain en un temps compris dans l'intervalle $[a, b]$. Ensuite, chaque occurrence est traitée instantanément par l'actionneur. L'estampille de la $n^{\text{ième}}$ action relative à la variable v en sortie de l'actionneur A est notée $t_n^{v_A}$. L'actionneur est relié à un équipement $e \quad Modules \quad RDC$. Les estampilles des événements de v en entrée du bus de terrain sont donc notées $t_n^{v_e}$. On a donc les contraintes suivantes entre ces estampilles :

$$n \quad \mathbb{N}\ ,\ t_n^{v_e} + a \leq t_n^{v_A} \leq t_n^{v_e} + b \tag{5.48}$$

Le processus d'un tel actionneur est alors :

$$P_A = \mathbf{s} \quad S^N \quad n \quad \mathbb{N}, \quad t_n^{v_e} + a \leq t_n^{v_A} \leq t_n^{v_e} + b \quad v_n^{v_A}{}_{v_e} = n \tag{5.49}$$

5.2.7 Remote Data Concentrator

Un RDC est une passerelle entre un ensemble de capteurs et d'actionneurs et le réseau AFDX. Les variables en entrée du RDC correspondent aux variables produites par les capteurs et celles arrivant du réseau AFDX, à destination des actionneurs. La figure 5.13 illustre le processus d'un RDC. Le processus P_R du RDC R reçoit les variables v et v du réseau. Comme un RDC est relié au réseau par un unique lien, ces variables ont traversé le même port du même commutateur pour rejoindre le RDC. Ce port est noté p et les signaux utilisés pour ces variables en entrée sont s_{v_p} et $s_{v'_p}$ et en sortie s_{v_R} et $s_{v'_R}$. Le RDC reçoit également les variables u et u en provenance des capteurs s et s respectivement. Pour ces variables, les signaux en entrée sont donc s_{u_s} et $s_{u'_s'}$ et en sortie s_{u_R} et $s_{u'_R}$.

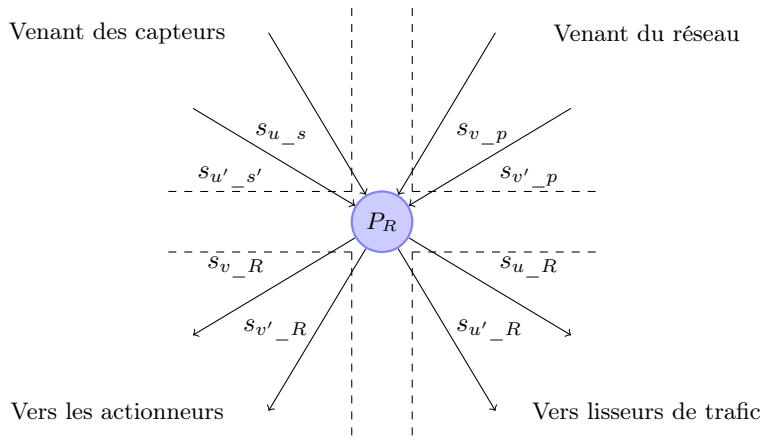


FIGURE 5.13 – Exemple du processus d'un RDC

Pour formaliser le processus associé à un RDC quelconque $R \quad RDC$, nous définissons deux ensembles de variables. Soit un RDC R, nous notons *R.InputSens* l'ensemble des variables en entrée arrivant d'un capteur :

$$R.InputSens = \quad v \quad Var \quad Alloc_S(v) = s \quad s.equipement = R \tag{5.50}$$

et *R.InputNet* l'ensemble des variables en entrée arrivant du réseau, à destination d'un actionneur :

$$R.InputNet = \quad v \quad Var \quad Alloc_P(v) = a \quad a.equipement = R \tag{5.51}$$

Le RDC s'active périodiquement avec une période égale à T_R et est sujet à un déphasage ϕ_R par rapport aux autres composants. Ainsi l'instant de la $n^{\text{ième}}$ activation du RDC est de la forme $\phi_R + n \cdot T_R$ avec $\phi_R \quad [0, T[$. Nous ne définissons pas d'événement particulier pour l'activation d'un RDC. Un RDC ne réalise que des encapsulations/désencapsulations, il n'y a donc pas de relations de dépendance entre variables dans le RDC : une occurrence d'une variable en sortie ne dépend que de la dernière occurrence de cette variable reçu en entrée. Plus formellement :

$$\begin{aligned} &v \quad R.InputSens \text{ , avec } Alloc_S(v) = s \text{ , } \quad n \quad \mathbb{N} \text{ , } v_n^{v_s}{}_x = \begin{cases} \max\ t_k^x \leq \phi_R + n \cdot T_R & \text{si } x = v_s \\ \text{sinon} \end{cases} \\ &v \quad R.InputNet \text{ , } \quad n \quad \mathbb{N} \text{ , } v_n^{v_p}{}_x = \begin{cases} \max\ t_k^x \leq \phi_R + n \cdot T_R & \text{si } x = v_p \\ \text{sinon} \end{cases} \end{aligned} \tag{5.52}$$

Pour ne pas encombrer la définition du processus, pour une variable $v \quad R.Input = R.InputSens$ $R.InpuNet$ nous notons E le dernier élément traversé par la variable avant d'arriver dans le RDC. Cet élément est soit un port du réseau soit un capteur. Dans tous les cas, chaque sortie est produite au pire après un temps C_R, donc chaque instant de production d'une variable v en sortie de R est de la forme : $\phi_R + n \cdot T_R + \delta$ avec $\delta \quad [0, C_R]$.Le processus d'un RDC R est alors :

$$\begin{aligned} P_R = \ \mathbf{s} \quad S^N \quad \phi_R \quad [0, T_R[, \quad v \quad R.Input, \quad n \quad \mathbb{N}\ , \\ \phi_R + n \cdot T_R \leq t_{n^-}^{v_R} \leq \phi_R + n \cdot T_R + C_R, \\ v_{n^-}^{v_E}{}_x = \begin{cases} \max\ t_k^x \leq \phi_R + n \cdot T_R & \text{si } x = v_E \\ \text{sinon} \end{cases} \end{aligned} \tag{5.53}$$

5.3 Comportements du système complet

Dans la section précédente, nous avons défini les comportements possibles de chacun des processus composant le système. Comme tous ces processus sont définis sur la même base de signaux, nous pouvons décrire l'ensemble des comportements possibles du système comme l'intersection des comportements de chacun de ses composants. Nous notons $\mathbb{S}$ l'ensemble des comportements possibles du système et nous avons :

$$\mathbb{S} = \bigcap_{m \ Modules} P_m \setminus \bigcap_{f \ Func} P_f \setminus \bigcap_{vl \ VL} P_{vl} \setminus \bigcap_{p \ Ports} P_p \setminus \bigcap_{s \ Sensors} P_s \setminus \bigcap_{a \ Actuators} P_a \setminus \bigcap_{r \ RDC} P_r$$

Pour rappel, cet ensemble de comportement est tel que $\mathbb{S} \subseteq S^N$ avec N le nombre de signaux utilisés dans la modélisation (c.f. section 5.1 page 75).

Ayant défini l'ensemble des comportements possibles d'un système, nous pouvons désormais formaliser les exigences temporelles que doit satisfaire le système.

Chapitre 6

Exigences

Dans ce chapitre, nous formalisons les exigences devant être respectées par le système. Plusieurs types d'exigences temporelles sont considérées : la latence, la fraîcheur et la cohérence. La définition des exigences s'appuie sur la notion de chaîne fonctionnelle. Une chaîne fonctionnelle est une séquence de fonctions qui partagent des variables. Par exemple, une exigence de latence concerne le temps écoulé entre la production d'une variable en entrée d'une chaîne et la production d'une variable dépendante en sortie de la chaîne. Nous considérons à la fois les exigences de latence, fraîcheur et cohérence dans le pire cas et dans le meilleur cas.

La sémantique des exigences est exprimée à l'aide de la modélisation dans le *tagged signal model* des comportements possibles du système.

6.1 Dé nitions préliminaires

6.1.1 Chaîne fonctionnelle

Une chaîne fonctionnelle $\mathcal{C}$ est une séquence de fonctions échangeant des variables telle que :

$$\mathcal{C} = {}^{v_0}\ F_1\ {}^{v_1}\ F_2 \ldots {}^{v_{n-1}}\ F_n\ {}^{v_n} \tag{6.1}$$

avec :

- i, $F_i \quad Func$,
- $i = 0..n-1$, $v_i \quad F_{i+1}.Input$,
- $i = 1..n$, $v_i \quad F_i.Output$,

Nous limitons notre étude à l'analyse de chaîne fonctionnelle sans boucle, c'est-à-dire que chaque variable n'apparaît qu'une seule fois dans la chaîne. En revanche, une chaîne fonctionnelle peut contenir plusieurs fois la même fonction.

Exemple 10 (Chaînes fonctionnelles de l'étude de cas)**.** *Pour illustrer la définition d'une chaîne fonctionnelle, nous définissons celles de l'étude de cas.*

La chaîne $\mathcal{F}_1$ définit la séquence de traitement : transformation de la mesure de la pression atmosphérique ($pres_1$) dans la centrale inertielle (fonction $ADIRU_1$) en une information de vitesse ($speed_1$), utilisation de la vitesse dans FM_1 pour calculer l'heure estimée d'arrivée (ETA_1), génération de la page à a cher ($disp_1$) par le gestionnaire d'a chage (MFD_1).

$$\mathcal{F}_1 = {}^{pres_1}\ ADIRU_1\ {}^{speed_1}\ FM_1\ {}^{ETA_1}\ MFD_1\ {}^{disp_1} \tag{6.2}$$

La chaîne $\mathcal{F}_2$ définit une séquence de traitement similaire à l'exception que la pression est mesurée par un capteur di érent et transformée par $ADIRU_2$:

$$\mathcal{F}_2 = {}^{pres_2}\ ADIRU_2\ {}^{speed_2}\ FM_1\ {}^{ETA_1}\ MFD_1\ {}^{disp_1} \tag{6.3}$$

La chaîne $\mathcal{L}_1$ correspond à la séquence de traitement mise en uvre entre la requête du pilote (req_1) et l'a chage des informations du waypoint *choisi sur son écran ($disp_1$) :*

$$\mathcal{L}_1 = {}^{req_1}\ KU_1\ {}^{wpId_1}\ FM_1\ {}^{query_1}\ NDB\ {}^{answer_1}\ FM_1\ {}^{wpInfo_1}\ MFD_1\ {}^{disp_1} \tag{6.4}$$

La chaîne $\mathcal{L}_2$ correspond à la séquence de traitement mise en uvre entre la requête du pilote (req_1) et l'a chage des informations du waypoint *choisi sur l'écran du copilote ($disp_2$) :*

$$\mathcal{L}_2 = {}^{req_1}\ KU_1\ {}^{wpId_1}\ FM_2\ {}^{query_2}\ NDB\ {}^{answer_2}\ FM_2\ {}^{wpInfo_2}\ MFD_2\ {}^{disp_2} \tag{6.5}$$

Les chaînes $\mathcal{F}_1$ et $\mathcal{F}_2$ servent à définir des exigences de fraîcheur et de cohérence. Les chaînes $\mathcal{L}_1$ et $\mathcal{L}_2$ servent à définir des exigences de latence et de cohérence.

6.1.2 Dépendance étendue

La formalisation des comportements du système dans le *tagged signal model* permet d'encoder les dépendances entre événements : un événement $s_j[n\ ]$ dépend de $s_i[n]$ ssi $s_j[n\ ]_i = n$, s_i et s_j étant des signaux en relation dans un même processus. Cependant, les dépendances dont nous avons besoin pour définir la sémantique des exigences concernent des événements répartis sur tout le système : pour définir une latence, il faut mesurer le temps écoulé entre un événement en entrée d'une chaîne fonctionnelle et l'événement en sortie qui dépend de lui.

Nous étendons la définition de dépendance entre événements pour prendre en compte les dépendances qui sont propagées entre événements.

Dé nition 15 (Dépendances étendues)**.** *On note la relation de dépendance étendue. Soit $s_i[n]$, $s_j[n\ ]$ deux événements d'un comportement $(s_1, \ldots, s_N)$ S^N. $s_j[n\ ]$ dépend de $s_i[n]$ ssi il existe une chaîne d'événements dépendants deux à deux qui relie $s_i[n]$ à $s_j[n\ ]$. On note alors $s_i[n]$ $s_j[n\ ]$.*

$$\begin{aligned} s_i[n] \quad s_j[n\ ] \qquad & m \quad \mathbb{N}, \\ & (k_1, \ldots, k_m) \quad (1..N)^m, k_1 = i \quad k_m = j \\ & (n_1, \ldots, n_m) \quad \mathbb{N}^m, n_1 = n \quad n_m = n \\ & l \quad (1..m-1), s_{k_{l+1}}[n_{l+1}]_{k_l} = n_l \end{aligned} \tag{6.6}$$

La relation de dépendance entre événements est transitive :

$$s_i[n] \quad s_j[n\ ] \text{ et } s_j[n\ ] \quad s_k[n\ ] = \quad s_i[n] \quad s_k[n\ ] \tag{6.7}$$

S'il n'existe pas de dépendance entre une paire d'événements $s_i[n]$ et $s_j[n\ ]$, alors nous notons :

$$s_i[n] \nrightarrow s_j[n\ ] \tag{6.8}$$

Exemple 11 (Dépendances étendues)**.** *Pour illustrer la notion de dépendances étendues entre signaux, nous représentons sur la figure 6.1 les premiers événements des signaux s_1, s_2 et s_3. Nous ne précisons pas ici les estampilles des événements, seules les valeurs des événements sont représentées.*

Les èches en trait plein sont les dépendances déduites de la valeur des événements. Par exemple, $s_2[3]$ dépend de $s_1[2]$ car la première composante de la valeur de $s_2[3]$ ($v^2_3{}_1$) est égale à 2. Comme $s_3[3]$ dépend de $s_2[3]$, par transitivité nous avons $s_3[3]$ dépend de $s_1[2]$ ($s_1[2]$ $s_3[3]$). Sur cet exemple nous avons également $s_1[1]$ $s_3[1]$, $s_1[1]$ $s_3[2]$ et $s_1[2]$ $s_3[4]$. Ces dépendances étendues (par transitivité) sont représentées en trait discontinu.

6.2 Exigences

La sémantique des exigences est exprimée dans le *tagged signal model*. L'idée est de tracer les dépendances entre événements en entrée et sortie de chaîne fonctionnelle par des dépendances étendues.

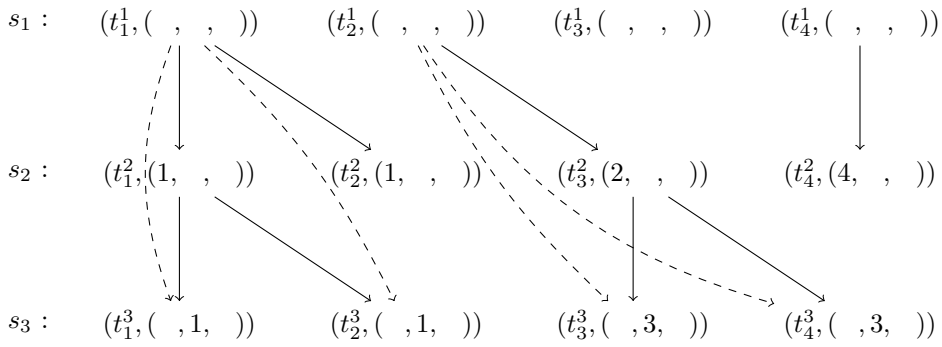


FIGURE 6.1 – Exemple de dépendances étendues

6.2.1 Latence

Latence pire cas

Une exigence de latence pire cas est utilisée pour garantir la réactivité du système vis-à-vis d'une chaîne fonctionnelle. La latence correspond au temps écoulé entre une action en entrée de la chaîne fonctionnelle et la première action en sortie qui en dépend. Soit une chaîne fonctionnelle $\mathcal{C} = {}^{v_0}\ F_1\ {}^{v_1}\ F_2 \ldots {}^{v_{n-1}}\ F_n\ {}^{v_n}$. On note S le capteur produisant la variable v_0 ($Alloc_S(v_0) = S$) et A l'actuateur utilisant v_n ($Alloc_P(v_n) = A$). Pour donner l'intuition de la latence, nous représentons les premiers événements d'un comportement possible du système sur la figure 6.2 qui ne contient que les événements des signaux en entrée de la chaîne ($s_{v_0_S}$) et en sortie ($s_{v_n_A}$). Dans les événements, nous n'indiquons que les estampilles des événements, les dépendances entre les occurrences de v_0 et v_n sont précisées à l'aide des flèches en trait discontinu. Sur cet exemple, la première occurrence de v_0 est utilisée par la première et la deuxième occurrences de v_n. La latence est une mesure du temps de réponse de la chaîne fonctionnelle, la latence de la première occurrence de v_0 est donc définie vis-à-vis de la première occurrence qui dépend d'elle. Cette latence est notée l_1 sur la figure et $l_1 = t_1^{v_n_A} - t_1^{v_0_S} = 40 - 20 = 20$ (l'unité de temps n'a pas d'importance pour cet exemple). Le temps écoulé vis-à-vis de la deuxième occurrence (et des suivantes si elles existaient) intervient dans la définition de la fraîcheur (c.f. sous section 6.2.2). Aucune occurrence de v_n ne dépend de la deuxième occurrence de v_0 donc pour elle la latence n'est pas définie. La première utilisation de la troisième occurrence de v_0 correspond à la troisième occurrence de v_n, on a alors une latence : $l_3 = t_3^{v_n_A} - t_3^{v_0_S} = 180 - 110 = 70$. Vérifier une exigence de latence pire cas consiste donc à s'assurer que pour toute occurrence en entrée de la chaîne, la latence ne dépasse pas une certaine borne.

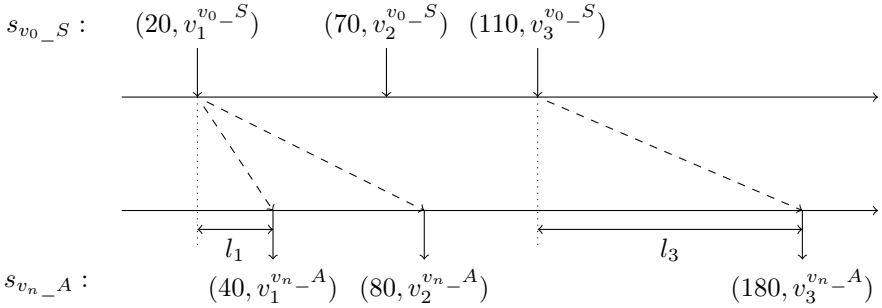


FIGURE 6.2 – Exemple de latences d'une chaîne

Plus formellement, nous notons une exigence de latence pire cas pour une chaîne fonctionnelle $\mathcal{C}$ et une borne maximale acceptable Λ :

$$Latency_{\leq \Lambda}(\mathcal{C}) \tag{6.9}$$

Une exigence de latence pire cas est satisfaite si la première utilisation en sortie de la chaîne de toute occurrence en entrée se fait en un temps inférieur à Λ. La satisfaction de l'exigence de latence pire cas est formalisée de la façon suivante :

$$\begin{aligned} \mathbb{S} = \mathit{Latency}_{\leq\Lambda}(\mathcal{C}) \qquad & \mathbf{s} \quad \mathbb{S}, \\ & i \quad \mathbb{N} \text{ t.q. } \quad j \quad \mathbb{N} \text{ avec } s_{v_0_S}[i] \quad s_{v_n_A}[j], \\ & j = \min \ k \quad \mathbb{N} \quad s_{v_0_S}[i] \quad s_{v_n_A}[k] \\ & t_j^{v_n_A} - t_i^{v_0_S} \leq \Lambda \end{aligned} \tag{6.10}$$

avec :
- $\mathbb{S}$ l'ensemble des comportements possibles du système,
- v_n la variable en sortie de la chaîne $\mathcal{C}$,
- A l'actionneur utilisant v_n,
- v_0 la variable en entrée de la chaîne $\mathcal{C}$,
- S le capteur produisant v_0,
- $\min \ k \quad \mathbb{N} \quad s_{v_0_S}[i] \quad s_{v_n_A}[k]$ l'indice du premier événement de v_n dépendant du $i^{\text{ème}}$ événement de v_0.

Outre la vérification de l'exigence, il peut être intéressant de connaître la valeur pire cas de la latence d'une chaîne fonctionnelle. Pour une chaîne $\mathcal{C}$ nous définissons $\mathit{WCL}(\mathcal{C})$ (WCL pour *Worst Case Latency*) telle que :

$$\mathit{WCL}(\mathcal{C}) = \max_{\mathbf{s} \ \mathbb{S}} \left\{ \max_{\substack{(i,j) \in \mathbb{N}^2 \\ s_{v_0_S}[i] \rightarrow s_{v_n_A}[j] \\ s_{v_0_S}[i] \nrightarrow s_{v_n_A}[j-1]}} t_j^{v_n_A} - t_i^{v_0_S} \right\} \tag{6.11}$$

La pire latence d'une chaîne est définie en utilisant la propriété suivante : si $s_{v_n_A}[j]$ dépend de $s_{v_0_S}[i]$ mais $s_{v_n_A}[j-1]$ ne dépend pas de $s_{v_0_S}[i]$ alors $s_{v_n_A}[j]$ est le premier événement qui dépend de $s_{v_0_S}[i]$.

Remarque : pour les mesures de latence, nous ne considérons pas les événements qui n'aboutissent pas. Ces événements n'influencent pas la sortie de la chaîne car ils sont écrasés par des événements plus récents en entrée de la chaîne. Pour le concepteur du système, ceci implique d'autres exigences qui seraient intéressantes à vérifier. Par exemple, il serait possible de vérifier que le système est dimensionné pour qu'aucun événement ne soit écrasé ($i \quad \mathbb{N}, \ j \quad \mathbb{N}$ t.q. $s_{v_0_S}[i] \quad s_{v_n_A}[j]$). Ou encore estimer le temps minimal entre deux événements pour que ces événements soient visibles en sortie (ce qui revient à déterminer $\min_{i>i'}(t_i^{v_0_S} - t_{i'}^{v_0_S})$ tel que $(j, j \ ) \quad \mathbb{N}$ avec $s_{v_0_S}[i] \quad s_{v_n_A}[j]$ et $s_{v_0_S}[i \ ] \quad s_{v_n_A}[j \ ]$).

Latence meilleure cas

Une exigence de latence meilleur cas permet de garantir que le temps écoulé entre un événement en entrée de la chaîne et un événement dépendant en sortie est toujours supérieur à une borne acceptable. Autrement dit, une exigence de latence meilleure cas garantit que le système n'est pas trop rapide. En première approche, une telle exigence peut sembler superflue dans la mesure où le concepteur du système devrait chercher à obtenir un système aussi rapide que possible. Mais pour certaines chaînes fonctionnelles, il est nécessaire de garantir que la variabilité (ou la gigue) de la latence n'est pas trop importante. Ceci permet d'assurer la régularité de la chaîne. Si une chaîne satisfait une exigence de latence pire cas avec une borne Λ et une exigence de latence meilleur cas avec une borne λ alors la chaîne à une gigue nécessairement inférieure à $\Lambda - \lambda$.

Une exigence de latence meilleur cas pour une chaîne fonctionnelle $\mathcal{C}$ et une borne minimale acceptable λ est notée :

$$\mathit{Latency}_{\geq\lambda}(\mathcal{C}) \tag{6.12}$$

Une exigence de latence meilleur cas est satisfaite si la première utilisation en sortie de la chaîne de toute occurrence en entrée se fait en un temps supérieur à λ :

$$\begin{aligned} \mathbb{S} = \mathit{Latency}_{\geq\lambda}(\mathcal{C}) \qquad & \mathbf{s} \quad \mathbb{S}, \\ & i \quad \mathbb{N} \text{ t.q. } \; j \quad \mathbb{N} \text{ avec } s_{v_0_S}[i] \quad s_{v_n_A}[j], \\ & j = \min \; k \quad \mathbb{N} \quad s_{v_0_S}[i] \quad s_{v_n_A}[k] \\ & t_j^{v_n_A} - t_i^{v_0_S} \geq \lambda \end{aligned} \tag{6.13}$$

avec :
- $\mathbb{S}$ l'ensemble des comportements possibles du système,
- v_n la variable en sortie de la chaîne $\mathcal{C}$,
- A l'actionneur utilisant v_n,
- v_0 la variable en entrée de la chaîne $\mathcal{C}$,
- S le capteur produisant v_0,
- $\min \; k \quad \mathbb{N} \quad s_{v_0_S}[i] \quad s_{v_n_A}[k]$ l'indice du premier événement de v_n dépendant du $i^{\text{ème}}$ événement de v_0.

Comme pour le pire cas, nous définissons le meilleur cas de latence pour une chaîne fonctionnelle $\mathcal{C}$, avec $BCL(\mathcal{C})$ (BCL pour *Best Case Latency*) :

$$BCL(\mathcal{C}) = \min_{\mathbf{s} \quad \mathbb{S}} \left\{ \begin{array}{c} \min \\ (i,j) \in \mathbb{N}^2 \\ s_{v_0}[i] \rightarrow s_{v_n}[j] \\ s_{v_0}[i] \nrightarrow s_{v_n}[j-1] \end{array} \quad t_j^{v_n_A} - t_i^{v_0_S} \right\} \tag{6.14}$$

6.2.2 Fraîcheur

Une exigence de fraîcheur permet de garantir que tout événement en sortie d'une chaîne fonctionnelle dépend d'événements suffisamment récents pour être pertinents. Nous reprenons l'exemple de la figure 6.2, pour représenter sur la figure 6.3 la fraîcheur des événements en sortie de la chaîne. Dans les événements, nous n'indiquons que les estampilles des événements, les dépendances entre les occurrences de v_0 et v_n sont précisées à l'aide des flèches en trait discontinu. Sur cet exemple, la première occurrence de v_n utilise la première occurrence de v_0. La fraîcheur de la première occurrence de v_n correspond donc au temps écoulé depuis la production de la première occurrence de v_0 et sa propre production. Cette fraîcheur est notée f_1 sur la figure et $f_1 = t_1^{v_n_A} - t_1^{v_0_S} = 40 - 20 = 20$. La fraîcheur de la deuxième occurrence de v_n est $f_2 = t_2^{v_n_A} - t_1^{v_0_S} = 80 - 20 = 60$.

Vérifier une exigence de fraîcheur pire cas consiste donc à s'assurer que pour toute occurrence en sortie de la chaîne, la fraîcheur ne dépasse pas une certaine borne.

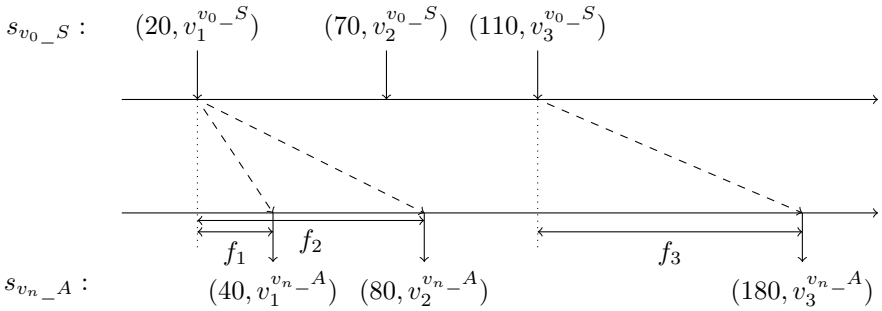


FIGURE 6.3 – Exemple de fraîcheurs d'une chaîne

Plus formellement, nous notons une exigence de fraîcheur pire cas pour une chaîne fonctionnelle $\mathcal{C}$ et une borne maximale acceptable Λ :

$$Freshness_{\leq \Lambda}(\mathcal{C}) \tag{6.15}$$

La satisfaction de l'exigence de latence pire cas est formalisée de la façon suivante :

$$\begin{aligned} \mathbb{S} = Freshness_{\leq \Lambda}(\mathcal{C}) \qquad & \mathbf{s} \quad \mathbb{S}, \\ & (i,j) \quad \mathbb{N}^2 \text{ t.q. } s_{v_0_S}[i] \quad s_{v_n_A}[j], \\ & t_j^{v_n_A} - t_i^{v_0_S} \leq \Lambda \end{aligned} \tag{6.16}$$

avec :
- $\mathbb{S}$ l'ensemble des comportements possibles du système,
- v_n la variable en sortie de la chaîne $\mathcal{C}$,
- A l'actionneur utilisant v_n,
- v_0 la variable en entrée de la chaîne $\mathcal{C}$,
- S le capteur produisant v_0.

Ainsi, et alors que la latence considère uniquement la première utilisation d'une occurrence, la fraîcheur pire cas considère toutes les utilisations et en particulier la dernière. Comme pour la latence, nous définissons la fraîcheur pire cas d'une chaîne fonctionnelle $\mathcal{C}$ avec $WCF(\mathcal{C})$ (WCF pour *Worst Case Freshness*) telle que :

$$WCF(\mathcal{C}) = \max_{\mathbf{s} \quad \mathbb{S}} \left\{ \max_{\substack{(i,j) \in \mathbb{N}^2 \\ s_{v_0_S}[i] \to s_{v_n_A}[j]}} t_j^{v_n_A} - t_i^{v_0_S} \right\} \tag{6.17}$$

Remarques :
- Nous ne définissons pas la notion d'exigence de fraîcheur meilleur cas dans la mesure où elle serait équivalente à une latence meilleur cas.
- On a toujours $WCL(\mathcal{C}) \leq WCF(\mathcal{C})$.

6.2.3 Cohérence

La notion d'exigence de cohérence permet d'offrir deux types de garantie :
- un même événement en entrée de plusieurs chaînes fonctionnelles doit résulter en des événements en sortie arrivant dans une même fenêtre temporelle. Par exemple, dans l'étude de cas, lorsque le pilote demande des informations sur un *waypoint*, ces informations doivent s'afficher simultanément (avec une certaine tolérance) sur les écrans du pilote et du copilote.
- un événement commun en sortie de plusieurs chaînes fonctionnelles doit dépendre d'événements en entrée de ces chaînes arrivant dans une même fenêtre temporelle. Par exemple, dans l'étude de cas, l'heure d'arrivée à un *waypoint* est estimée à l'aide des données inertielles issues de deux capteurs différents. Pour que cette estimation ait du sens, il est nécessaire de garantir que les données utilisées sont mesurées dans une même fenêtre temporelle.

L'expression de ces exigences de cohérence est donc formalisée sur des ensembles de chaînes fonctionnelles. Nous distinguerons ces deux catégories d'exigences de cohérence avec les notions de chaînes fonctionnelles convergentes et de chaînes fonctionnelles divergentes. Pour des raisons de clarté, nous définissons ces notions pour des ensembles de deux chaînes fonctionnelles. L'extension à des ensembles de n chaînes fonctionnelles ne pose pas de difficulté technique.

Exigence de cohérence sur chaînes fonctionnelles divergentes

Pour commencer, nous précisons la notion de chaînes divergentes. Des chaînes fonctionnelles sont dites divergentes si elles partagent toutes une même variable en entrée. Soit deux chaînes fonctionnelles $\mathcal{C} = {}^{v_0} F_1 {}^{v_1} F_2 \ldots {}^{v_{n-1}} F_n {}^{v_n}$ et $\mathcal{C} = {}^{v'_0} F_1 {}^{v'_1} F_2 \ldots {}^{v'_{n'-1}} F_{n'} {}^{v'_{n'}}$, $\mathcal{C}$ et $\mathcal{C}$ sont dites divergentes si et seulement si $v_0 = v_0$.

Remarques : des chaînes fonctionnelles divergentes n'ont pas nécessairement la même longueur, et ne divergent pas nécessairement après leur première fonction, comme représenté sur la figure 6.4. Bien que partageant une même première variable, des chaînes fonctionnelles divergentes ne partagent pas nécessairement une même première fonction, comme illustré sur la figure 6.5.

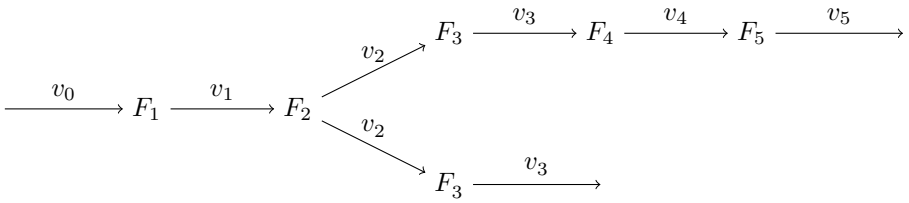


FIGURE 6.4 – Exemple de chaînes divergentes

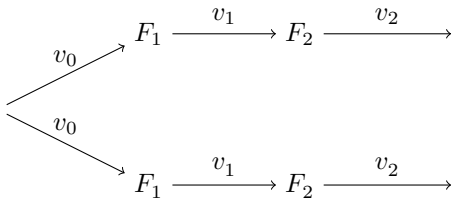


FIGURE 6.5 – Exemple de chaînes divergentes avec $F_1 = F_1$

Exemple 12 (Chaînes de traitement d'une requête du pilote)**.** *Pour illustrer la notion de chaîne fonctionnelles divergentes, nous utilisons les deux chaînes fonctionnelles impliquées dans le traitement d'une requête du pilote : $\mathcal{L}_1$ et $\mathcal{L}_2$. Chacune des chaînes correspond à une séquence de traitement produisant l'a chage des informations du* waypoint *sélectionné par le pilote sur un des écrans. Ces deux chaînes sont représentées sur la figure 6.6.*

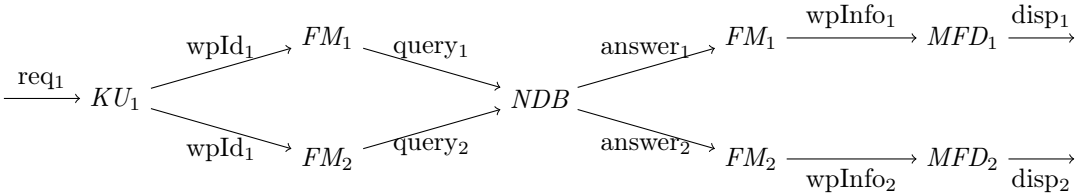


FIGURE 6.6 – Exemple de chaînes divergentes de l'étude de cas

L'exigence de cohérence sur ces chaînes fonctionnelles sert à garantir que les a chages sur les écrans du pilote (disp_1) et du copilote (disp_2) sont cohérents : les deux écrans doivent a cher les mêmes informations, en même temps, à la suite d'une saisie du pilote. Le système exécutant étant par nature asynchrone, il n'est pas possible de garantir que ces a chages seront simultanés. Une exigence est alors définie par rapport à une borne maximale, c'est-à-dire que le temps écoulé entre ces a chages ne doit jamais dépasser cette borne.

Sur la figure 6.7, le pilote envoie deux requêtes (A) et (B) ; c_A et c_B représentent la cohérence entre les chaînes fonctionnelles pour ces deux requêtes.

Formalisation. Nous notons une exigence de cohérence pour deux chaînes fonctionnelles divergentes $\mathcal{C}$ et $\mathcal{C}$, une borne maximale acceptable Λ (nous utilisons le terme *ConsistD* pour *Consistency of Divergent functional chains*) :

$$ConsistD_{\leq\Lambda}(\mathcal{C}, \mathcal{C}\ ) \tag{6.18}$$

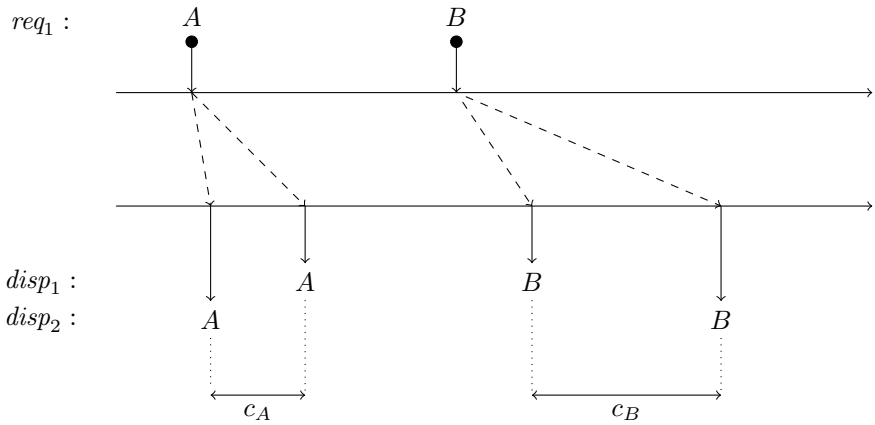


FIGURE 6.7 – Illustration de la cohérence sur chaînes fonctionnelles divergentes

La satisfaction de l'exigence de cohérence est alors formalisée de la façon suivante :

$$
\begin{aligned}
\mathbb{S} = \mathit{ConsistD}_{\leq \Lambda}(\mathcal{C}, \mathcal{C}\ ) \qquad & \mathbf{s} \quad \mathbb{S}, \\
& (i, j, k) \quad \mathbb{N}^3, \\
& s_{v_0_S}[i] \quad s_{v_n_A}[j], \\
& j = \min\ l \quad \mathbb{N} \quad s_{v_0_S}[i] \quad s_{v_n_A}[l]\ , \\
& s_{v_0_S}[i] \quad s_{v'_{n'}_A'}[k], \\
& k = \min\ l \quad \mathbb{N} \quad s_{v_0_S}[i] \quad s_{v'_{n'}_A'}[l]\ , \\
& t_j^{v_n_A} - t_k^{v'_{n'}_A'} \leq \Lambda
\end{aligned}
\tag{6.19}
$$

avec :

- $\mathbb{S}$ l'ensemble des comportements possibles du système,
- v_n la variable en sortie de la chaîne $\mathcal{C}$,
- A l'actionneur utilisant v_n,
- $v_{n'}$ la variable en sortie de la chaîne $\mathcal{C}$,
- A le capteur utilisant $v_{n'}$,
- v_0 la variable commune en entrée des chaînes $\mathcal{C}$ et $\mathcal{C}$,
- S le capteur produisant v_0.

Comme pour la latence et la fraîcheur, nous exprimons le pire cas de cohérence pour des chaînes fonctionnelles divergentes $\mathcal{C}$ et $\mathcal{C}$ avec $\mathit{WCCD}(\mathcal{C}, \mathcal{C}\ )$ (WCCD pour *Worst Case Consistency of Divergent functional chains*) tel que :

$$
\mathit{WCCD}(\mathcal{C}, \mathcal{C}\ ) = \max_{\mathbf{s}\ \ \mathbb{S}} \left\{ \begin{array}{c} \max \\ (i, j, k) \in \mathbb{N}^3, \\ s_{v_0_S}[i] \rightarrow s_{v_n_A}[j], j = \min\{l \in \mathbb{N} \mid s_{v_0_S}[i] \rightarrow s_{v_n_A}[l]\}, \\ s_{v_0_S}[i] \rightarrow s_{v'_{n'}_A'}[k], k = \min\{l \in \mathbb{N} \mid s_{v_0_S}[i] \rightarrow s_{v'_{n'}}\} \end{array} \quad t_j^{v_n_A} - t_k^{v'_{n'}_A'} \right\}
\tag{6.20}
$$

Le meilleur cas de cette cohérence est défini de manière similaire en remplaçant les "max" par des "min" dans la définition de *WCCD*. Ce meilleur cas est alors noté $\mathit{BCCD}(\mathcal{C}, \mathcal{C}\ )$ (BCCD pour *Best Case Consistency of Divergent functional chains*).

Exigence de cohérence sur chaînes fonctionnelles convergentes

Des chaînes fonctionnelles sont dites convergentes si elles partagent toutes une même variable en sortie. Soient deux chaînes fonctionnelles $\mathcal{C} = {}^{v_0} F_1 {}^{v_1} F_2 \ldots {}^{v_{n-1}} F_n {}^{v_n}$ et $\mathcal{C} = {}^{v'_0} F_1 {}^{v'_1} F_2 \ldots {}^{v'_{n'-1}} F_{n'} {}^{v'_{n'}}$, $\mathcal{C}$ et $\mathcal{C}$ sont dites convergentes si et seulement si $v_n = v_{n'}$. Par conséquent, on a nécessairement $F_n = F_{n'}$.

Des chaînes fonctionnelles convergentes n'ont pas nécessairement la même longueur, et ne convergent pas nécessairement juste avant leur dernière fonction, comme représenté sur la figure 6.8 :

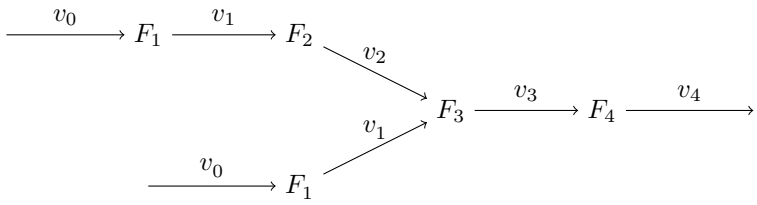


FIGURE 6.8 – Exemple de chaînes convergentes

Exemple 13 (Chaînes de traitement pour l'affichage de l'heure estimée d'arrivée)**.** *Pour illustrer la notion de chaînes fonctionnelles convergentes, nous utilisons les deux chaînes fonctionnelles impliquées dans l'estimation de l'heure d'arrivée à un* waypoint*. Chacune des chaînes correspond à une séquence de traitement transformant une mesure physique (la pression) en une donnée inertielle (vitesse) puis en heure estimée d'arrivée. Ces deux chaînes sont représentées sur la figure 6.9 :*

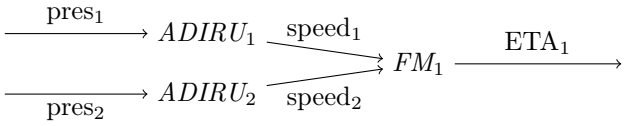


FIGURE 6.9 – Exemple de chaînes convergentes de l'étude de cas

Désormais, nous pouvons formaliser la notion d'exigence de cohérence sur des chaînes fonctionnelles convergentes.

Pour illustrer une exigence de cohérence sur des chaînes convergentes, considérons deux chaînes convergentes $\mathcal{C}$ et $\mathcal{C}$ telles que présentées dans la définition. La variable v_0 en entrée de $\mathcal{C}$ est produite par le capteur S et la variable v_0 en entrée de $\mathcal{C}$ est produite par le capteur S . Nous représentons plusieurs mesures de la cohérence sur la figure 6.2.3. Seules les estampilles des événements sont indiquées. Les deux chaînes convergent vers une variable commune v_n en sortie, utilisée dans l'actionneur A. Ainsi, chaque occurrence de v_n dépend d'une occurrence de v_0 et d'une occurrence de v_0. Les flèches en trait discontinu symbolisent ces dépendances. La première occurrence de v_n dépend des premières occurrences de v_0 et v_0. La cohérence vaut alors $c_1 = t_1^{v_0-S} - t_1^{v'_0-S'} = 20 - 40 = 20$. Pour la deuxième occurrence de v_n, v_0 n'a pas été mise à jour donc c'est encore la première occurrence de v_0 qui est utilisée, et la cohérence est moins bonne : $c_2 = t_2^{v_0-S} - t_1^{v'_0-S'} = 100 - 40 = 60$. La cohérence pour la troisième occurrence est $c_3 = t_2^{v_0-S} - t_2^{v'_0-S'} = 100 - 150 = 50$.

Nous notons une exigence de cohérence pour deux chaînes fonctionnelles convergentes $\mathcal{C}$, $\mathcal{C}$ et une borne maximale acceptable Λ (nous utilisons le terme *ConsistC* pour *Consistency of Convergent functional chains*) :

$$ConsistC_{\leq\Lambda}(\mathcal{C}, \mathcal{C}) \tag{6.21}$$

La satisfaction de l'exigence de cohérence est alors formalisée de la façon suivante :

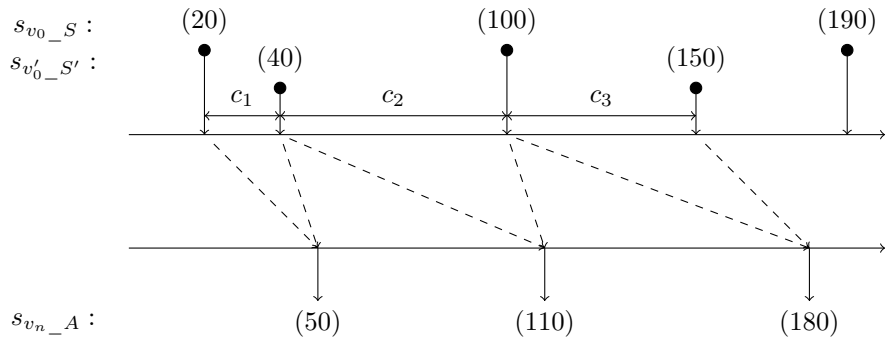


FIGURE 6.10 – Exemple de mesures de cohérence pour deux chaînes convergentes

$$\begin{aligned} \mathbb{S} = \mathit{ConsistC}_{\leq\Lambda}(\mathcal{C},\mathcal{C}\ ) \qquad & \mathbf{s} \quad \mathbb{S}, \\ & (i,j,k) \quad \mathbb{N}^3, \\ & s_{v_0_S}[i] \quad s_{v_n_A}[k], \\ & s_{v'_0_S'}[j] \quad s_{v_n_A}[k], \\ & t_i^{v_0_S} - t_j^{v'_0_S'} \ \leq \Lambda \end{aligned} \tag{6.22}$$

avec :
- $\mathbb{S}$ l'ensemble des comportements possibles du système,
- v_0 la variable en entrée de la chaîne $\mathcal{C}$,
- S le capteur produisant v_0,
- v_0 la variable en entrée de la chaîne $\mathcal{C}$,
- S le capteur produisant v_0,
- v_n la variable commune en sortie des chaînes $\mathcal{C}$ et $\mathcal{C}$,
- A l'actionneur utilisant v_n.

Comme pour la latence et la fraîcheur, nous exprimons le pire cas de cohérence pour des chaînes fonctionnelles convergentes $\mathcal{C}$ et $\mathcal{C}$ avec $\mathit{WCCC}(\mathcal{C},\mathcal{C}\ )$ (WCCC pour *Worst Case Consistency of Convergent functional chains*) tel que :

$$\mathit{WCCC}(\mathcal{C},\mathcal{C}\ ) = \max_{\mathbf{s}\ \ \mathbb{S}} \left\{ \max_{\substack{\forall(i,j,k)\in\mathbb{N}^3, \\ s_{v_0_S}[i] \to s_{v_n_A}[k], \\ s_{v'_0_S'}[j] \to s_{v_n_A}[k]}} t_i^{v_0_S} - t_j^{v'_0_S'} \right\} \tag{6.23}$$

Le meilleur cas de cette cohérence est défini de manière similaire en remplaçant les "max" par des "min" dans la définition de *WCCC*. Ce meilleur cas est alors noté $\mathit{BCCC}(\mathcal{C},\mathcal{C}\ )$ (BCCC pour *Best Case Consistency of Convergent functional chains*).

6.2.4 Exigences de l'étude de cas

Nous résumons dans cette sous-section les exigences de l'étude de cas qui s'appuient sur les chaînes fonctionnelles décrites dans l'exemple 10 (page 93). Les différentes bornes sont exprimées en milliseconde.

Exigences de latence. Deux exigences de latence sont définies, la première sur la chaîne $\mathcal{L}_1$ et la seconde sur $\mathcal{L}_2$:

$$\mathit{Latency}_{\leq 700}(\mathcal{L}_1) \text{ et } \mathit{Latency}_{\leq 700}(\mathcal{L}_2) \tag{6.24}$$

Exigences de fraîcheur. Deux exigences de fraîcheur sont définies, la première sur la chaîne $\mathcal{F}_1$ et la seconde sur $\mathcal{F}_2$:

$$Freshness_{\leq 400}(\mathcal{F}_1) \text{ et } Freshness_{\leq 400}(\mathcal{F}_2) \tag{6.25}$$

Exigences de cohérence entre chaînes fonctionnelles divergentes. Une exigence de cohérence est définie entre les chaînes fonctionnelles divergentes $\mathcal{L}_1$ et $\mathcal{L}_2$:

$$ConsistD_{\leq 500}(\mathcal{L}_1, \mathcal{L}_2) \tag{6.26}$$

Exigences de cohérence entre chaînes fonctionnelles convergentes. Une exigence de cohérence est définie entre les chaînes fonctionnelles convergentes $\mathcal{F}_1$ et $\mathcal{F}_2$:

$$ConsistC_{\leq 300}(\mathcal{F}_1, \mathcal{F}_2) \tag{6.27}$$

6.3 Conclusion

Ce chapitre termine la partie II consacrée à la modélisation d'un système IMA. Dans un premier temps, au chapitre 4, nous avons introduit un modèle d'architecture composé de trois éléments : la description de l'architecture fonctionnelle du système, la description de son architecture matérielle et l'allocation de l'architecture fonctionnelle sur l'architecture matérielle. Un ensemble de propriétés a également été défini. Le respect de ces propriétés permet de garantir que le système est structurellement bien formé, par exemple qu'il existe un chemin dans le réseau reliant la source à la destination d'une variable. Ce modèle ne décrit pas le comportement temporel des composants du système, il ne permet donc pas de vérifier la correction du système vis-à-vis d'exigences temps réel. Nous avons donc défini, au chapitre 5, un modèle de comportement fondé sur le *tagged signal model*. Les comportements possibles d'un composant du système sont capturés au sein d'un processus, les comportements possibles du système global sont déterminés par la composition de l'ensemble des processus de ses composants. Les exigences temps réel devant être satisfaites par le système, c'est-à-dire les exigences de latence, de fraîcheur et de cohérence, ont alors été formalisées dans le *tagged signal model*. Chaque exigence est définie sur une ou plusieurs chaînes fonctionnelles qui représentent des séquences de fonctions. Nous avons également décrit dans ce formalisme les propriétés meilleur et pire cas associées à ces exigences. La prochaine partie est dédiée au calcul des meilleurs et pires cas permettant la vérification des exigences.

Troisième partie

Véri cation des exigences

Introduction de la troisième partie

Maintenant que la sémantique du système et des exigences est définie, nous pouvons l'exploiter pour vérifier la satisfaction des exigences temps réel. L'idée de notre approche est la suivante : nous choisissons un ensemble d'événements quelconques d'un comportement du système, directement impliqués dans la (ou les) chaîne(s) fonctionnelle(s) d'une exigence. Nous listons alors l'ensemble des contraintes que doivent satisfaire ces événements. Toute solution du système de contraintes ainsi défini, appartient alors à un comportement du système. Avec le problème ainsi formalisé (chapitre 7), il est possible d'explorer les comportements du système relatifs à une exigence particulière. Une approche par abstraction est introduite (section 7.1) pour limiter le nombre de contraintes d'un tel système. Ensuite (chapitre 8), pour déterminer le pire ou meilleur cas de l'exigence, ce système de contraintes est utilisé dans un solveur de programmes linéaires. Pour ce faire, nous linéarisons certaines contraintes et nous exprimons un pire ou meilleur cas comme une fonction objectif d'un programme linéaire mixte. Cette méthode est testée sur les exigences de l'étude de cas. Les résultats obtenus sont comparés à des formules analytiques *locales*, c'est-à-dire considérant que le pire cas d'une exigence est obtenu en additionnant les pires cas de chaque élément le long d'une chaîne fonctionnelle par exemple.

Chapitre 7

Formalisation du problème

Dans ce chapitre, nous présentons la formalisation du problème sous une forme permettant la vérification de la satisfaction d'une exigence temps réel. Dans un premier temps (section 7.1), nous présentons la démarche suivie pour abstraire une partie du système. En particulier, les processus concrets des lisseurs de trafic et du réseau sont remplacés par des processus abstraits qui constituent une sur-approximation de leurs comportements. Ceci permet de casser la complexité du modèle et rend possible la vérification des exigences. Ensuite (section 7.2), à partir de ce modèle abstrait, pour chaque exigence, l'idée est de constituer un système de contraintes associé à l'ensemble de ses chaînes fonctionnelles. Ce système de contraintes contient alors toutes les contraintes reliant un événement en entrée d'une chaîne à un événement dépendant en sortie. Toute solution à ce système de contraintes définit alors un ensemble d'événements appartenant à un comportement possible du système abstrait. Comme le système abstrait est une sur-approximation, toute solution valide dans ce système l'est également dans le système concret. Il est ainsi possible d'analyser les propriétés temporelles de la chaîne fonctionnelle. Finalement (section 7.3), nous décrivons la mise en œuvre pratique de cette approche au sein de notre prototype.

7.1 Abstraction du problème

L'expression exacte de l'ensemble des comportements possibles du système est difficilement exploitable pour analyser un système de grande taille. En effet, la complexité d'une telle analyse serait trop importante. Classiquement, des techniques d'abstraction sont utilisées pour réduire la complexité. Nous décrivons dans cette partie notre démarche d'abstraction ainsi que les méthodes utilisées pour construire ces abstractions. Trois caractéristiques du modèle expliquant la complexité peuvent être identifiées :

- l'asynchronisme des modules,
- la variabilité des temps d'exécution des fonctions,
- les contentions non-déterministes dans le réseau inter-module dues à l'asynchronisme des communications.

Une approche classique d'abstraction est de considérer un réseau de communication comme un ensemble de canaux temporisés [34, 35]. Chaque canal représente alors un lien entre un unique émetteur et un unique récepteur, et ce lien impose un délai compris dans un intervalle borné à chaque message échangé entre l'émetteur et le récepteur. Pour garantir que cette abstraction ait du sens, il faut prouver que l'intervalle caractérisant un canal temporisé est correct : tout échange entre l'émetteur et le récepteur doit effectivement s'effectuer dans un temps compris dans cet intervalle. Des techniques d'analyse réseau pire cas, comme le *Network Calculus* ou l'*approche par trajectoire*, peuvent être utilisées pour ces preuves.

7.1.1 Principes généraux

Considérons un système concret $\mathbb{S}_C$ et une exigence Φ. Nous devons donc prouver que :

$$\mathbb{S}_C = \Phi \tag{7.1}$$

Une exigence peut être :

- *universelle* : une certaine propriété doit être vérifiée pour *tous* les comportements possibles du système,

– *existentielle* : une certaine propriété doit être vérifiée pour *au moins un* des comportements possibles du système.

Nous nous intéressons uniquement à des exigences de type universel : quelque soit le comportement du système, les exigences de latence, de fraîcheur ou de cohérence doivent être satisfaites. Plus formellement,

$$\mathbb{S}_C = \Phi \qquad \mathbf{s} \quad \mathbb{S}_C \ , \ \mathbf{s} = \Phi \tag{7.2}$$

L'approche par abstraction est la suivante : nous définissons un système abstrait tel que celui-ci soit une sur-approximation du système concret. Nous avons donc une relation $\mathbb{S}_C \subseteq \mathbb{S}_A$. Nous savons alors que si le système abstrait satisfait l'exigence, alors le système concret la satisfait aussi. En effet, si nous prouvons que $\mathbb{S}_A \supseteq \mathbb{S}_C$ alors nous prouvons directement que l'implication $\mathbb{S}_A = \Phi \ = \ \mathbb{S}_C = \Phi$ est correcte. Supposons que la proposition $\mathbb{S}_A = \Phi$ soit vraie. Si Φ est une exigence universelle, on a alors :

$$\mathbf{s} \quad \mathbb{S}_A \ , \ \mathbf{s} = \Phi \tag{7.3}$$

Si $\mathbb{S}_A \supseteq \mathbb{S}_C$, alors tous les comportements du système concret sont inclus dans les comportements du système abstrait et donc : $\mathbf{s} \quad \mathbb{S}_C \ , \ \mathbf{s} = \Phi$. Ce qui est équivalent à $\mathbb{S}_C = \Phi$. L'implication est donc vérifiée.

A noter que, comme le système abstrait peut contenir des comportements qui n'appartiennent pas au système concret, si une exigence n'est pas satisfaite sur le système abstrait, elle peut l'être sur le système concret.

Remarquons également que si nous nous intéressions à des exigences existentielles, les abstractions utilisées devraient être telles que $\mathbb{S}_C \supseteq \mathbb{S}_A$. En effet, contrairement aux exigences universelles, une exigence existentielle est satisfaite si il *existe* au moins un comportement possible vérifiant une certaine propriété. Ainsi une abstraction ne peut être utilisée que si l'ensemble de ces comportements est inclus dans les comportements possibles du système concret.

Deux étapes sont donc nécessaires pour vérifier qu'un système satisfait une exigence :

1. la définition d'une abstraction du système concret et la preuve que : $\mathbb{S}_C \subseteq \mathbb{S}_A$,
2. la preuve que le système abstrait satisfait l'exigence à vérifier : $\mathbb{S}_A = \Phi$

7.1.2 Particularités du *tagged signal model*

La définition d'un système abstrait dans le *tagged signal model* se fait en remplaçant un ou plusieurs processus concrets par un ou plusieurs processus abstraits, incluant l'ensemble des comportements des processus remplacés. Nous présentons dans un premier temps le remplacement d'un unique processus par un processus abstrait. Ensuite, nous étendrons cette idée au cas d'une abstraction remplaçant un ensemble de processus concrets par un ensemble de processus abstraits.

Remplacement d un processus par un processus abstrait

Nous illustrons cette idée sur la figure 7.1. Dans cet exemple, $\mathbb{S}_C$ est un système concret composé des processus P_1, P_2 et P_3. On a alors $\mathbb{S}_C = P_1 \setminus P_2 \setminus P_3$. Nous construisons un système abstrait $\mathbb{S}_A$ tel que $\mathbb{S}_A = P_1 \setminus A_2 \setminus P_3$ avec A_2 un processus abstrayant le comportement de P_2.

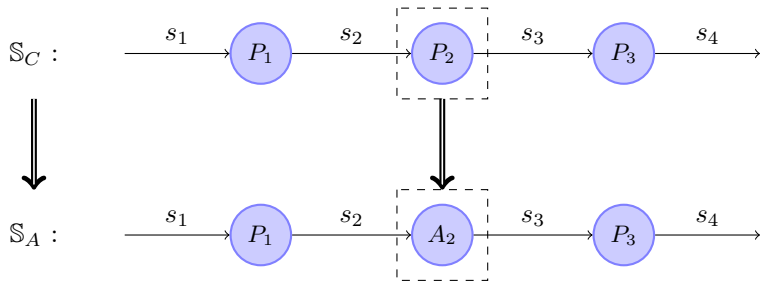


FIGURE 7.1 – Abstraction d'un processus dans le *tagged signal model*

Comme vu précédemment, cette abstraction est correcte vis-à-vis de nos objectifs (vérification d'exigences universelles) si et seulement si $\mathbb{S}_A \supseteq \mathbb{S}_C$. Nous remarquons alors que :

$$\begin{aligned} &\text{si } A_2 \supseteq P_2 \\ &\text{alors } P_1 \setminus A_2 \supseteq P_1 \setminus P_2 \\ &\text{et } P_1 \setminus A_2 \setminus P_3 \supseteq P_1 \setminus P_2 \setminus P_3 \\ &\text{donc } \mathbb{S}_A \supseteq \mathbb{S}_C \end{aligned} \tag{7.4}$$

Ainsi, si nous prouvons que $A_2 \supseteq P_2$, alors l'abstraction est utilisable pour la vérification. Remarquons que même si le processus A_2 n'inclut pas tous les comportements de P_2 ($A_2 \supseteq P_2$), il peut constituer une abstraction correcte. En effet, si il est possible de prouver $P_1 \setminus A_2 \supseteq P_1 \setminus P_2$ ou $A_2 \setminus P_3 \supseteq P_2 \setminus P_3$, alors on peut également en déduire que $P_1 \setminus A_2 \setminus P_3 \supseteq P_1 \setminus P_2 \setminus P_3$ et donc que l'abstraction est utilisable. Dans la suite, nous proposons d'abstraire le processus P_{VL} d'un lisseur de trafic par un processus A_{VL} tel que $A_{\mathrm{VL}} \supseteq P_{VL}$ (c.f. section 7.1.3 page 111). Mais, la composition de ce processus abstrait avec le processus de la fonction f du lisseur du trafic est telle que $P_f \setminus A_{VL} \supseteq P_f \setminus P_{VL}$. Ainsi, cette abstraction est acceptable.

Remplacement d un ensemble de processus par un ensemble de processus abstraits

Prenons maintenant l'exemple de l'abstraction d'un processus par un autre ensemble de processus représenté sur la figure 7.2. C'est ce que nous utiliserons pour définir une abstraction du réseau AFDX. Dans cet exemple, le système concret est $\mathbb{S}_C = P_1 \setminus P_2 \setminus P_3 \setminus P_4$. Le système abstrait est $\mathbb{S}_A = P_1 \setminus A_1 \setminus A_2$. Il est intéressant de remarquer que les signaux $s_c, s_{c'}, s_d, s_{d'}$ ne sont plus liés à des processus dans le système abstrait. Ils sont « libres » et ne sont soumis à aucune contrainte. Ces signaux font néanmoins partie de la définition du système. C'est-à-dire qu'un comportement du système abstrait $\mathbf{s} \quad \mathbb{S}_A$ est de la forme $\mathbf{s} = (s_a, s_b, s_{b'}, s_c, s_{c'}, s_d, s_{d'}, s_e, s_{e'})$. Il est nécessaire que $\mathbb{S}_C$ et $\mathbb{S}_A$ soient toujours définis sur une même base de signaux, sinon la comparaison de ces ensembles n'aurait plus de sens car on aurait alors nécessairement $\mathbb{S}_A \supseteq \mathbb{S}_C$. Ou alors la comparaison serait plus complexe à définir car il faudrait utiliser une projection sur le sous-ensemble des signaux communs aux deux systèmes (les fonctions de projection utilisées par Lee et Sangiovanni-Vincentelli sont décrites dans [76, 75]). Naturellement, les exigences à vérifier ne doivent pas porter sur les signaux qui ne sont plus liés à des processus dans le système abstrait.

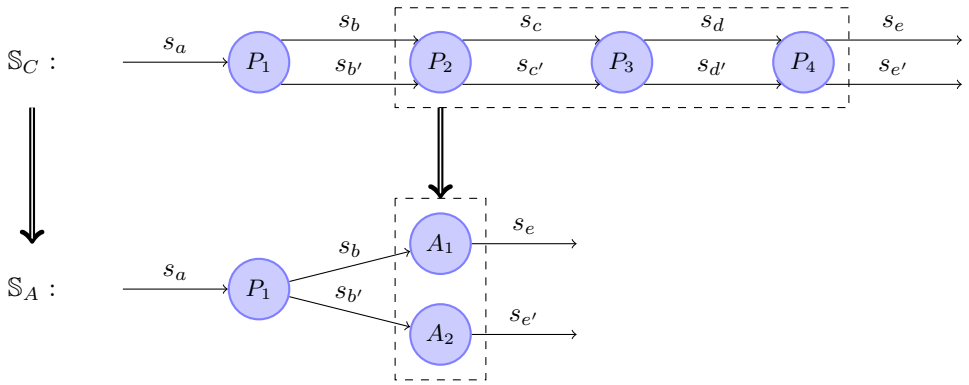


FIGURE 7.2 – Abstraction d'un ensemble de processus par un autre ensemble de processus

Comme pour le cas du remplacement d'un unique processus, pour montrer que l'abstraction est correcte, il faut alors prouver soit que $A_1 \setminus A_2 \supseteq P_2 \setminus P_3 \setminus P_4$ ou directement que $P_1 \setminus A_1 \setminus A_2 \supseteq P_1 \setminus P_2 \setminus P_3 \setminus P_4$.

7.1.3 Abstraction des lisseurs de trafic

Pour simplifier l'expression des comportements des lisseurs de trafic, nous utilisons trois hypothèses :

1. une fonction produit toutes ses sorties au même instant,

2. le lisseur de trafic transmet toutes les trames produites lors d'une exécution avant l'exécution suivante,
3. le nombre maximal de trames envoyées à un lisseur de trafic est connu, quelle que soit l'exécution.

Nous définissons donc un processus abstrait pour un lisseur de trafic qui prend en compte certaines spécificités de sa fonction émettrice. Cette démarche est valable car dans les systèmes étudiés les entrées d'un lisseur de trafic sont nécessairement des sorties d'une fonction. A noter qu'un lisseur de trafic peut également être en sortie d'un RDC. Dans ce cas, les mêmes hypothèses et la même démarche d'abstraction sont appliquées.

Dé nition de l abstraction d un lisseur de tra c

Nous définissons un processus qui abstrait le processus concret d'un lisseur de trafic tel que décrit dans la formule 5.34 (page 85). Le processus abstrait d'un lisseur de trafic VL, utilisé par les occurrences des variables $v_1, \ldots, v_l$, produites par la fonction f est :

$$\begin{aligned} A_{VL} = \ \mathbf{s} \quad S^N \quad & i = 1..l, \quad n \quad \mathbb{N}, \quad c = 0..N_{VL} - 1 \text{ t.q.} \\ & t_n^{v_i_VL} = t_n^{v_i_f} + c \cdot bag, \\ & v_n^{v_i_VL}{}_{v_i_f} = n, \\ & j = 1..l, \quad m \quad \mathbb{N} \text{ t.q. } (j,m) = (i,n) \\ & t_m^{v_j_VL} = t_n^{v_i_VL} \end{aligned} \tag{7.5}$$

avec N_{VL} le nombre maximal d'occurrences de variables délivées au lisseur de trafic lors d'une exécution de f. Au niveau du système, l'abstraction revient à remplacer le processus concret du lisseur de trafic P_{VL} par le processus abstrait A_{VL}.

Propriété 1 (Correction du processus abstrait d'un lisseur de trafic)**.** *Pour toute fonction f Func modélisée par le processus concret P_f, émettant des variables au travers d'un lien virtuel VL dont le lisseur de trafic est modélisé par le processus concret P_{VL} et est abstrait par le processus abstrait A_{VL}, nous avons l'inclusion suivante :*

$$A_{VL} \setminus P_f \supseteq P_{VL} \setminus P_f \tag{7.6}$$

Cette inclusion garantit la correction de l'abstraction du lisseur de trafic.

Preuve de la correction de l abstraction

Pour simplifier la définition des comportements possibles d'un lisseur de trafic (c.f. formule 5.34 page 85), nous utilisons l'hypothèse qu'une fonction produit toutes ses variables en sortie au même instant et celle précisant que le lisseur de trafic transmet toutes les trames produites lors d'une exécution avant l'exécution suivante. Cette dernière hypothèse garantit que la capacité de la file d'attente du lisseur de trafic n'est jamais dépassée.

Soit une fonction f transmettant les variables $v_1, \ldots, v_l$ au travers du lisseur de trafic du lien virtuel VL. Comme décrit dans la définition du processus concret P_{VL} d'un lisseur de trafic, les dates d'arrivée et de sortie de la $n^{\text{ième}}$ occurrence variable v_i sont liées de la façon suivante :

$$t_n^{v_i_VL} = t_n^{v_i_f} + W(t_n^{v_i_f}) + I(v_i, n) \tag{7.7}$$

avec

- $W(t_n^{v_f})$ la charge de travail restant avant l'arrivée de l'occurrence de la variable et générée par les occurrences arrivées strictement avant $t_n^{v_f}$,
- $I(v, n)$ le retard dû aux occurrences arrivées exactement à $t_n^{v_f}$ et traitées avant l'occurrence n de v.

Le processus A_{VL} n est pas une abstraction correcte. Pour cette abstraction, nous n'avons pas l'inclusion du processus concret dans le processus abstrait, c'est-à-dire $A_{VL} \supseteq P_{VL}$. En effet, considérons un VL avec deux variables v_1 et v_2 issues de la fonction f en entrée. Nous exhibons un comportement de P_{VL} n'appartenant pas à son abstraction A_{VL}.

Soit un comportement $\mathbf{s} \quad P_{VL}$ tel que l'ensemble des signaux soient vides, à l'exception des signaux contraints par P_{VL} : $s_{v_1_f}, s_{v_2_f}, s_{v_1_VL}$ et $s_{v_1_VL}$. Chacun de ces signaux contient un unique événement ($s_{v_1_f}[1]$ pour $s_{v_1_f}$ par exemple). Soit $\delta \quad]0, bag[$ tel que les estampilles des événements des variables venant de la fonction f soient distantes de δ :

$$t_1^{v_1-f} + \delta = t_1^{v_2-f} \tag{7.8}$$

Dans un premier temps, nous déterminons les estampilles des événements en sortie ($s_{v_1_VL}[1]$ et $s_{v_2_VL}[1]$) qui respectent les contraintes définissant le processus P_{VL}. Ensuite, nous montrerons que le comportement défini par ces estampilles ne peut pas appartenir au processus A_{VL}.

Pour l'estampille de l'événement de la variable v_1 en sortie, nous avons la relation :

$$t_1^{v_1-VL} = t_1^{v_1-f} + W(t_1^{v_1-f}) + I(v_1, 1) \tag{7.9}$$

$W(t_1^{v_1-f})$ est la charge de travail générée par les occurrences des variables arrivées strictement avant $t_1^{v_1-f}$. Comme $s_{v_1_f}[1]$ est le premier événement du comportement $\mathbf{s}$ (on a choisi $\delta > 0$) nécessairement $W(t_1^{v_1-f}) = 0$. En outre, comme $\delta > 0$, les deux événements $s_{v_1_f}[1]$ et $s_{v_1_f}[2]$ ne peuvent arriver simultanément et ainsi $I(v_1, 1) = 0$. Ainsi l'occurrence de la variable v_1 est traitée instantanément par le lisseur de trafic et $t_1^{v_1-VL} = t_1^{v_1-f}$.

Pour l'estampille de l'événement de la variable v_1 en sortie, nous avons la relation :

$$t_1^{v_2-VL} = t_1^{v_2-f} + W(t_1^{v_2-f}) + I(v_2, 1) \tag{7.10}$$

$W(t_1^{v_2-f})$ est la charge de travail générée par les occurrences des variables arrivées strictement avant $t_1^{v_2-f}$. Comme $s_{v_1_f}[1]$ est arrivée un temps $\delta < bag$ avant $s_{v_1_f}[1]$, il reste une charge de travail de $bag - \delta$. Comme pour $s_{v_1_f}[1]$ aucune occurrence ne peut arriver au même instant que $s_{v_1_f}[1]$ et donc $I(v_2, 1) = 0$. Ainsi, l'instant de sortie de l'occurrence de v_2 est nécessairement tel que :

$$t_1^{v_2-VL} = t_1^{v_2-f} + bag - \delta \tag{7.11}$$

Nous montrons à présent qu'un comportement avec ces estampilles ne peut appartenir à A_{VL}. Nous appliquons un raisonnement par l'absurde. Supposons que $\mathbf{s} \quad A_{VL}$. D'après les contraintes définissant le processus :

$$c = 0..N_{VL} - 1 \text{ t.q. } t_1^{v_2-VL} = t_1^{v_2-f} + c \cdot bag \tag{7.12}$$

Or comme nous l'avons vu $t_1^{v_2-VL} = t_1^{v_2-f} + bag - \delta$ avec $\delta \quad]0, bag[$ et donc il ne peut exister $c = 0..N_{VL}-1$ vérifiant la contrainte précédente. Il y a donc une contradiction et nous avons $\mathbf{s} \quad A_{VL}$. Ce qui montre que :

$$\begin{aligned} &\mathbf{s} \quad P_{VL} \text{ t.q. } \mathbf{s} \quad A_{VL} \\ &= \quad P_{VL} \subseteq A_{VL} \end{aligned} \tag{7.13}$$

Nous ne pouvons donc pas prouver directement la correction de l'abstraction proposée.

Prise en compte du processus P_f. Nous considérons alors les processus issus de la composition des processus concret et abstrait des lisseurs de trafic avec le processus de la fonction, c'est-à-dire : $P_{VL} \setminus P_f$ et $A_{VL} \setminus P_f$. Ainsi, nous pouvons prendre en compte les hypothèses faites sur les instants de production des variables par la fonction et celle stipulant qu'il ne reste aucune trame de l'exécution précédente.

Ces hypothèses sont valables quelque soit le type de la fonction, c'est-à-dire produisant des variables périodiques, sporadiques ou un mélange des deux.

Comme il ne peut rester aucune trame d'une exécution précédente de f dans le lisseur de trafic, seules les trames produites lors de l'exécution courante peuvent participer à $W(t_n^{v-f})$. Mais comme toutes les occurrences des variables sont produites au même instant, on a alors $W(t_n^{v-f}) = 0$.

La date de sortie du lisseur de trafic ne dépend donc que du nombre d'occurrences produites par la fonction lors de l'exécution. Avec cette hypothèse, l'expression de l'instant de sortie la $n^{\text{ième}}$ occurrence d'une variable n se réduit alors à :

$$t_n^{v-VL} = t_n^{v-f} + I(v, n) \tag{7.14}$$

Comme vu dans la description du processus d'un lisseur de trafic, le terme $I(v, n)$ représente le retard imposé par les occurrences produites par la fonction en même temps que l'occurrence n de la variable v. Pour un comportement donné, $I(v, n) = c \cdot bag$ avec c le nombre d'occurrences traitées avant l'occurrence n de la variable v et produites par f au même instant. La plage de valeurs que peut prendre c est comprise entre 0 (pas de retard, l'occurrence est traitée en premier par le lisseur de trafic) et $n_I(t_n^{v-f}) - 1$ avec $n_I(t_n^{v-f})$ le nombre d'occurrences produites à t_n^{v-f} (l'occurrence est traitée en dernier).

En considérant qu'à chaque exécution de f toutes les occurrences sont produites en même temps, alors $n_I(t_n^{v-f})$ représente le nombre d'occurrences produites par f lors de l'exécution produisant la $n^{\text{ième}}$ occurrence de la variable v. Le nombre d'occurrences produites peut varier d'une exécution à une autre si certaines des variables envoyées dans le lisseur de trafic sont des variables sporadiques. La plage de valeurs de c est donc variable, ce qui complique le modèle. Pour l'abstraction, nous cherchons à exprimer cette plage de valeurs avec des constantes. Nous connaissons le nombre maximal d'occurrences produites pour une exécution N_{VL} (c'est un paramètre fourni par le concepteur du système). Nous pouvons donc exprimer la plage de valeurs de c avec des bornes fixes : $c = 0..N_{VL} - 1$. Cela revient à considérer que le maximum de variables sporadiques est émis à chaque période de f.

Ainsi, nous prouvons la propriété $A_{VL} \setminus P_f \supseteq P_{VL} \setminus P_f$ et donc la correction du processus abstrait du lisseur de trafic.

7.1.4 Abstraction du réseau : canaux temporisés

On abstrait également les processus des ports des commutateurs et des ports des modules. En effet, l'évaluation des propriétés temporelles d'un réseau de file d'attente est connue pour être un problème complexe [27]. Nous ne pouvons donc pas prendre en compte les comportements exacts du réseau tels que nous les avons décrits au chapitre 5. Ainsi, nous décidons d'utiliser une sur-approximation des comportements du réseau, construite à l'aide de composants nommés *canaux temporisés*. Nous utilisons un canal temporisé pour chaque chemin de chaque lien virtuel. Un canal est caractérisé par un intervalle de temps $[a, b]$: a (respectivement b) étant la borne inférieure (respectivement supérieure) du temps de traversé du réseau pour une trame empruntant le lien virtuel correspondant au canal.

Les bornes des canaux temporisés peuvent être déterminées à l'aide des méthodes classiques d'évaluation de performance telles que le *Network Calculus* [31, 74, 52] ou l'*approche par trajectoire* [85, 84, 17, 18].

Abstraction

Pour l'abstraction du réseau, nous proposons de remplacer l'ensemble des processus des ports de communication $P_{p \quad p \quad Ports}$ par un ensemble de processus modélisant des canaux temporisés. Nous utilisons un canal temporisé pour chaque chemin de chaque lien virtuel. Le nombre de canaux temporisés utilisés N_C est donc la somme du nombre de chemins de chaque VL :

$$N_C = \sum_{vl \quad VL} card(vl.paths) \tag{7.15}$$

Nous illustrons le principe de cette abstraction sur une partie de l'étude de cas. La figure 7.3 représente l'architecture de cette étude de cas restreinte.

Sur la figure 7.4, nous représentons, dans la partie supérieure, une partie des processus impliqués dans l'étude de cas restreinte (les processus des modules n'apparaissent pas) ainsi que les processus du système

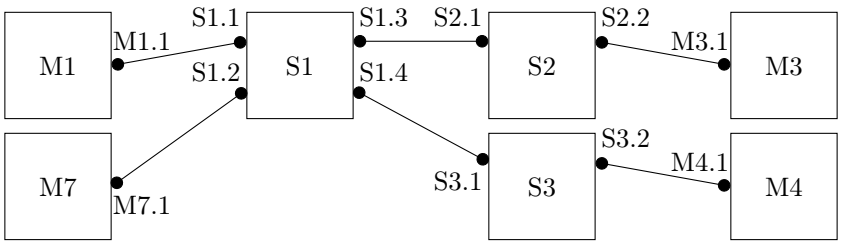


FIGURE 7.3 – Architecture de l'étude de cas restreinte

abstrait dans la partie inférieure. Les processus des ports de communication ($P_{M1.1}$, $P_{M7.1}$, $P_{S1.3}$, $P_{S1.4}$, $P_{S2.2}$ et $P_{S3.2}$) sont remplacés par des processus de canaux temporisés (P_{C_1}, P_{C_2}, P_{C_3} et P_{C_4}). Sur le système abstrait, nous utilisons des synonymes pour faciliter la lecture de la figure. Ainsi, le signal $s_{wp_S2.2}$ (en sortie de $P_{S2.2}$ portant la variable *wp*) du système concret devient $s_{wp_C_1}$ en sortie du processus du canal temporisé P_{C_1}. A noter que sur cette figure nous avons également appliqué l'abstraction des lisseurs de trafic, telle que présentée précédemment, en remplaçant les processus P_{VL_1}, P_{VL_2} et P_{VL_3} par A_{VL_1}, A_{VL_2} et A_{VL_3} respectivement.

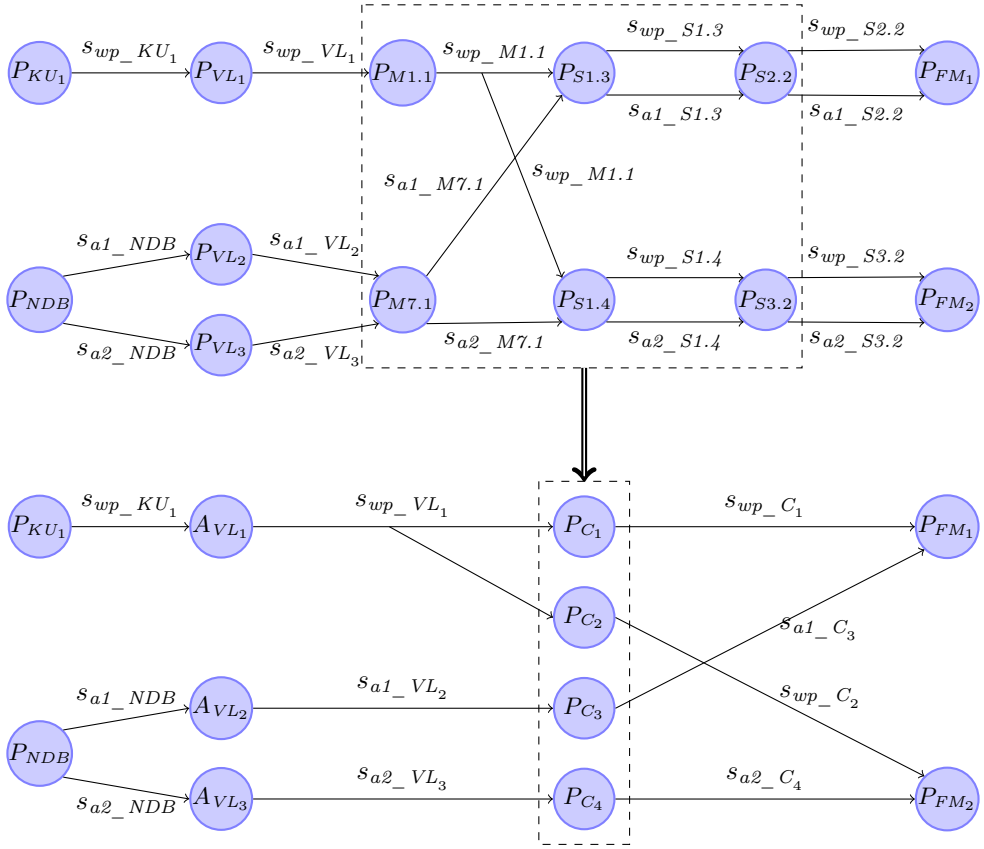


FIGURE 7.4 – Abstraction du réseau de l'étude de cas restreinte

Pour un lien virtuel *vl* par lequel transitent les variables $v_1, \ldots, v_l$ et un chemin p $vl.paths$ de ce VL, nous associons un canal temporisé C tel que :

$$\begin{aligned} P_C = \ \mathbf{s} \quad S^N \quad & i = 1..l, \ n \quad \mathbb{N}, \quad \delta \quad [a,b], \\ & t_n^{v_i_C} = t_n^{v_i_vl} + \delta \\ & v_n^{v_i_C}{}_{v_i_vl} = n \end{aligned} \tag{7.16}$$

avec
- $t_n^{v_i}{-}^{C}$ un synonyme de $t_n^{v_i}{-}^{last(p)}$ ($last(p)$ étant le dernier port sur le chemin p),
- a $\mathbb{N}$ une borne inférieure du temps de traversée du chemin p,
- b $\mathbb{N}$ une borne supérieure du temps de traversée du chemin p,

La borne inférieure a correspond au temps de traversée de la plus petite trame du lien virtuel parcourant le réseau, sans rencontrer d'autres trames dans les files d'attentes des ports. Le temps de traversée du réseau correspond donc à la somme des latences technologiques des commutateurs traversés et des temps d'émission de la trame en sortie de chacun des équipements le long du chemin. Pour un canal temporisé C correspondant à un chemin p d'un lien virtuel vl, la borne inférieure du temps de traversée a est déterminée par la formule suivante :

$$a = \sum_{s \quad S(p)} s.lag + \sum_{(l_{in}, l_{out}) \quad p} \frac{vl.s_{min}}{l_{in}.d} \tag{7.17}$$

avec $S(p)$ l'ensemble des commutateurs le long du chemin p et $s.lag$ la latence technologique du commutateur s (c.f. section 4.2 page 65).

La borne supérieure b est déterminée à l'aide de l'*approche par trajectoire* que nous avons présentée dans l'état de l'art (cf. sous-section 2.3.2 page 39).

Discussion

L'abstraction du réseau est une sur-approximation pour deux raisons. La première raison est que les techniques utilisées pour estimer les pires et meilleurs temps de traversée du réseau sont sur-approximatives par nature. Ainsi les bornes des canaux temporisés peuvent ne pas être atteintes par le système concret. Les sources de ce pessimisme sont étudiées dans [79].

Cependant, même si les bornes des canaux étaient atteignables par le système concret, l'abstraction reste source de sur-approximations. Prenons l'exemple d'une fonction émettant une variable u au travers de deux liens virtuels distincts. La fonction ne produit pas d'autres variables et à chaque exécution exactement une occurrence de u et de v sont produites. Lors d'une exécution de cette fonction, les occurrences de u et v arrivent ainsi simultanément dans le port de sortie du module. Le pire cas de traversée du port de sortie pour une occurrence de u survient quand l'occurrence de v est traitée juste avant elle dans le port de sortie. Une occurrence de u réalise son pire cas lors que l'occurrence de v réalise son meilleur cas. Les pires cas de traversée de u et v sont donc exclusifs dans le système concret. Si on construit un système abstrait en remplaçant le processus du port de sortie du module par deux canaux temporisés (un pour chaque lien virtuel), alors cette propriété ne peut pas être prise en compte car les canaux sont indépendants. Le système abstrait admet ainsi un ensemble plus large de comportements, ce qui en fait une sur-approximation du système concret.

Application à l étude de cas

En appliquant cette méthode de sur-approximation, l'architecture de l'étude de cas présentée sur la figure 1.4 (page 10) est remplacée par celle de la figure 7.5. Les ports de sortie des modules, des liens, des commutateurs et les liens vituels utilisant ces équipements sont remplacés par un ensemble de canaux temporisés. Par exemple, VL_1 qui a pour source KU_1 et pour destinations FM_1 et FM_2 est remplacé par les canaux temporisés C_1 et C_1.

Les caractéristiques de ces canaux sont résumées dans le tableau 7.1. La colonne BCTT correspond à la borne inférieure du temps de traversée (BCTT pour *Best Case Traversal Time*) et la colonne WCTT correspond à la borne supérieure du temps de traversée (WCTT pour *Worst Case Traversal Time*).

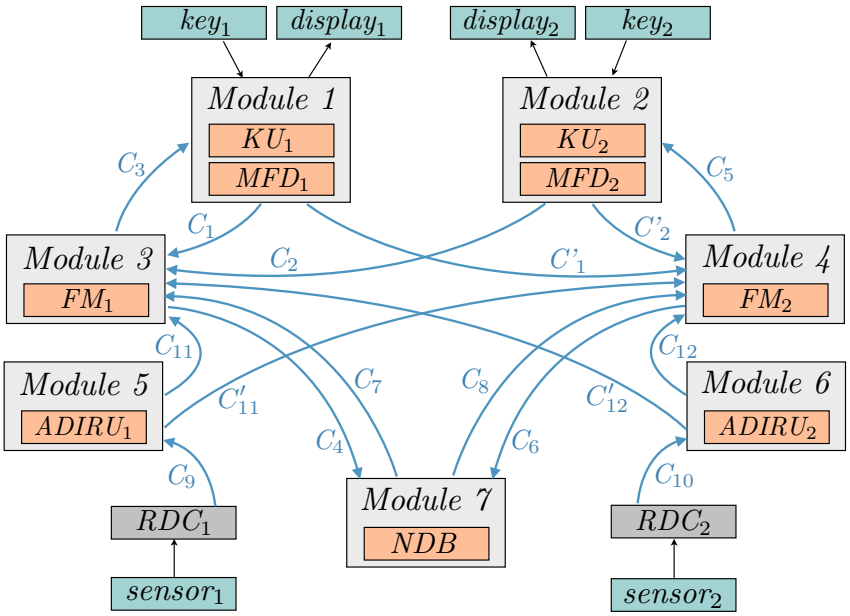


FIGURE 7.5 – Architecture abstraite de l'étude de cas

canal	source	destination	variable(s)	BCTT (μs)	WCTT (μs)
C_1	KU_1	FM_1	$wpId_1$	298	444
C'_1	KU_1	FM_2	$wpId_1$	298	444
C_2	KU_2	FM_1	$wpId_2$	298	444
C'_2	KU_2	FM_2	$wpId_2$	298	444
C_3	FM_1	MFD_1	$wpInfo_1$, ETA_1	310	490
C_4	FM_1	NDB	$query_1$	310	450
C_5	FM_2	MFD_2	$wpInfo_2$, ETA_2	310	490
C_6	FM_2	NDB	$query_2$	310	450
C_7	NDB	FM_1	$answer_1$	400	508
C_8	NDB	FM_2	$answer_2$	400	508
C_9	RDC_1	$ADIRU_1$	$pres_1$	150	156
C_{10}	RDC_2	$ADIRU_2$	$pres_2$	150	156
C_{11}	$ADIRU_1$	FM_1	$speed_1$	452	584
C'_{11}	$ADIRU_1$	FM_2	$speed_1$	452	584
C_{12}	$ADIRU_2$	FM_2	$speed_2$	452	584
C'_{12}	$ADIRU_2$	FM_1	$speed_2$	452	584

TABLE 7.1 – Caractéristiques des canaux temporisés de l'étude de cas

7.2 Intuitions pour la méthode de véri cation

A partir du modèle abstrait du système, développé dans la section précédente, il est envisageable d'énumérer l'ensemble des contraintes influençant certains événements du système, et ce pour tout comportement. Dans cette section, nous introduisons cette idée qui est utilisée pour la vérification des exigences temps réel. Prenons l'exemple d'une chaîne fonctionnelle $\mathcal{C} = {}^{x}\ f\ {}^{y}$ et supposons que cette chaîne fonctionnelle doit respecter une certaine exigence de fraîcheur : $Freshness_{\leq \Lambda}(\mathcal{C})$. Nous considérons ici le système représenté sur la figure 7.6. La variable x est fournie par le capteur S qui est directement relié au module M où s'exécute la fonction f. La variable y est utilisée par l'actionneur A qui est relié au module.

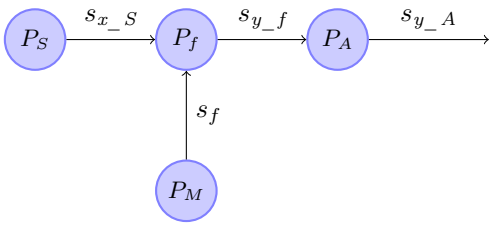


FIGURE 7.6 – Un système illustratif

Pour vérifier cette exigence, nous déterminons le pire cas de fraîcheur de cette chaîne qui est défini (c.f. sous-section 6.2.2 page 97) par :

$$WCF(\mathcal{C}) = \max_{\mathbf{s}\ \ \mathbb{S}} \left\{ \max_{\substack{(i,j) \in \mathbb{N}^2 \\ s_{x_S}[i] \to s_{y_A}[j]}} t_j^{y_A} - t_i^{x_S} \right\} \tag{7.18}$$

L'idée est alors de considérer un comportement $\mathbf{s}\ \ \mathbb{S}$ générique et un couple d'événements ($s_{x_S}[i]$, $s_{y_A}[j]$) tel que l'événement en sortie sur y ($s_{y_A}[j]$) dépend de l'événement en entrée sur x ($s_{x_S}[i]$). Cette dépendance se note $s_{x_S}[i]\ \ \ s_{y_A}[j]$. A partir de ces événements représentant la chaîne fonctionnelle à analyser, nous allons déterminer dans le système de contraintes défini dans le *tagged signal model* un sous-ensemble de contraintes. Les contraintes choisies sont celles qui influencent les événements de la chaîne fonctionnelle. Le choix des contraintes est ainsi guidé par la chaîne fonctionnelle étudiée. Connaissant ces contraintes nous pouvons déterminer le pire cas de fraîcheur (ou une borne supérieure). Comme le comportement et les événements considérés sont génériques, le pire cas de fraîcheur est alors valable pour tous les comportements et tous les couples d'événements.

On détaille ce principe sur l'exemple de la figure 7.6. Soit $\mathbf{s} = (s_{x_S}, s_{y_f}, s_{y_A}, s_f)\ \ \ \mathbb{S}$ et $(i,j)\ \ \ \mathbb{N}^2$ tels que $s_{x_S}[i]\ \ \ s_{y_A}[j]$. Cette dépendance implique qu'il existe une chaîne d'événements le long de la chaîne fonctionnelle liant $s_{x_S}[i]$ à $s_{y_A}[j]$. Plus formellement :

$$k\ \ \ \mathbb{N} \text{ t.q. } v_{y_A}[j]_{\ y_f} = k \text{ et } v_{y_f}[k]_{\ x_S} = i \tag{7.19}$$

Autrement dit, $s_{y_f}[k]$ est un événement sur la variable y en sortie de f qui dépend de $s_{x_S}[i]$ et dont dépend $s_{y_A}[j]$.

Pour analyser temporellement cette chaîne d'événements, nous listons l'ensemble des contraintes qu'elle doit satisfaire. L'événement $s_{x_S}[i]$ au début de cette chaîne est contraint dans le processus du capteur P_S. De ce processus, nous tirons donc la contrainte :

$$\phi_S\ \ \ [0, T_S[\text{ et } \ \delta_S\ \ \ [a_S, b_S] \text{ t.q. } t_i^{x_S} = \phi_S + i \cdot T_S + \delta_S \tag{7.20}$$

avec T_S la période du capteur, a_S (respectivement b_S) le meilleur (respectivement le pire) temps de traversée du lien reliant le capteur au module. Cet événement est également contraint par le processus P_f de la fonction f. En particulier, si $s_{y_f}[k]$ dépend de $s_{x_S}[i]$, alors nous avons :

$$i = \max\ k \quad \mathbb{N} \quad t_k^{x_S} \leq t_k^f \tag{7.21}$$

Cette contrainte n'est pas exploitable car elle lie le $k^{\text{ème}}$ instant d'activation de f à toutes les productions de x en sortie de S. On peut cependant l'interpréter de la façon suivante : $t_i^{x_S}$ correspond au dernier événement arrivé avant t_k^f si l'événement suivant arrive strictement après t_k^f. Autrement dit, pour définir la dépendance, on peut remplacer la contrainte précédente par les deux contraintes suivantes :

$$t_i^{x_S} \leq t_k^f \quad t_{i+1}^{x_S} > t_k^f \tag{7.22}$$

Nous devons donc prendre en compte dans le système de contraintes celles relatives à l'événement $s_{x_S}[i+1]$. Cet événement subit le même déphasage du capteur Φ_S que $s_{x_S}[i]$, mais peut traverser le bus de terrain en un temps différent, on a donc la contrainte suivante issue du processus P_S :

$$\delta_S \quad [a_S, b_S] \text{ t.q. } t_{i+1}^{x_S} = \phi_S + (i+1) \cdot T_S + \delta_S \tag{7.23}$$

L'événement d'activation de la fonction $s_f[k]$ apparaît en outre dans deux autres contraintes : l'une provient du processus du module P_M et l'autre du processus de la fonction P_f. En effet, l'estampille de l'événement $s_{y_f}[k]$ est contrainte par l'estampille de la $k^{\text{ième}}$ activation de f dans le processus P_f :

$$\delta_f \quad [0, C_f] \text{ t.q. } t_k^{y_f} = t_k^f + \delta_f \tag{7.24}$$

Et l'événement $s_f[k]$ est contraint dans le processus P_M par :

$$\phi_M \quad [0, HP(M)[\text{ t.q. } t_k^f = \phi_M + O_f + k \cdot T_f \tag{7.25}$$

Finalement les événements $s_{y_f}[k]$ et $s_{y_A}[j]$ sont liés dans le processus de l'actionneur :

$$k = j \quad t_k^{y_f} + a_A \leq t_j^{y_A} \quad t_j^{y_A} \leq t_k^{y_f} + b_A \tag{7.26}$$

Pour cet exemple, le système de contraintes engendré par la chaîne d'événements représentant la chaîne fonctionnelle $\mathcal{C}$ est alors :

$$\begin{cases} t_i^{x_S} = \phi_S + i \cdot T_S + \delta_S \\ t_i^{x_S} \leq t_k^f \\ t_{i+1}^{x_S} > t_k^f \\ t_{i+1}^{x_S} = \phi_S + (i+1) \cdot T_S + \delta_S \\ t_k^{y_f} = t_k^f + \delta_f \\ t_k^f = \phi_M + O_f + k \cdot T_f \\ t_k^{y_f} + a_A \leq t_j^{y_A} \\ t_j^{y_A} \leq t_k^{y_f} + b_A \\ k = j \end{cases} \tag{7.27}$$

Les variables continues de ce système sont les estampilles $t_i^{x_S}$, $t_{i+1}^{x_S}$, t_k^f, $t_k^{y_f}$ et $t_j^{y_A}$ ainsi que les phases ϕ_S et ϕ_M, et les temps de traversée ou d'exécution δ_S, δ_S et δ_f. Les variables discrètes sont les indices des événements i,k et j. Le système de contraintes relatif à l'exigence de latence de la chaîne fonctionnelle $\mathcal{L}_1$ est donné en exemple dans l'annexe B.

Nous pouvons représenter ce système de contraintes à l'aide d'un hypergraphe (un hypergraphe est un graphe dans lequel un arc peut relier un nombre quelconque de nœuds, et non pas un ou deux nœuds). Les nœuds sont alors les variables du système (estampilles des événements, phases des modules...) et les arcs représentent les contraintes entre des sous-ensembles de ces variables. Un tel hypergraphe est représenté sur la figure 7.7.

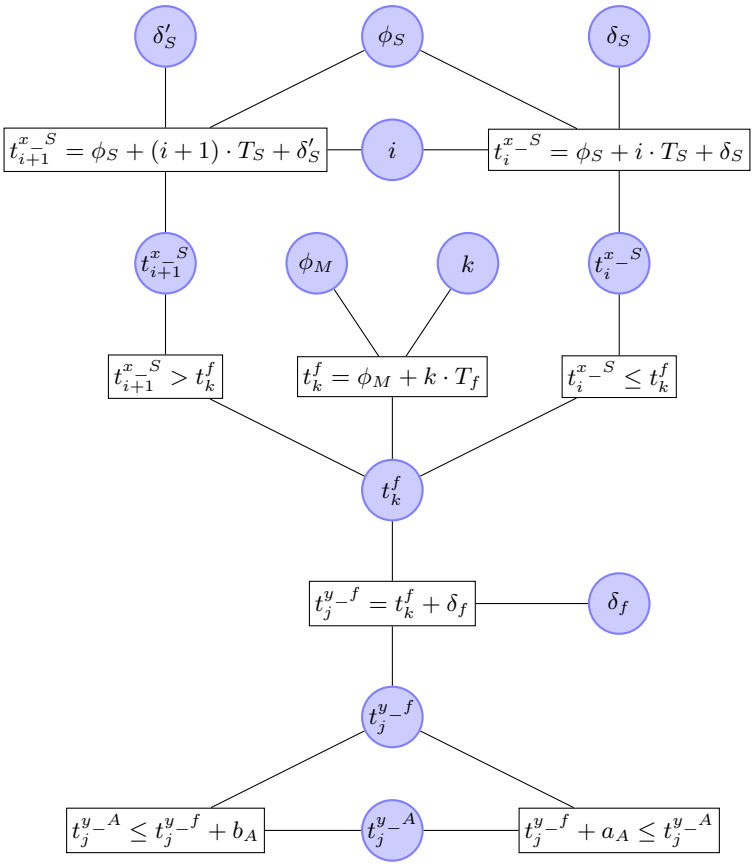


FIGURE 7.7 – Graphe des contraintes

Particularité de la latence. La fraîcheur est définie entre un événement en entrée d'une chaîne fonctionnelle et n'importe quel événement en sortie qui en dépend. Ainsi, lors du choix des événements à considérer dans le système de contraintes, il faut juste garantir que les occurrences de variables considérées ne sont pas écrasées par les occurrences suivantes. Dans l'exemple, la contrainte 7.22 est utilisée à cet effet.

De même, lors de la constitution du système de contraintes pour une exigence de latence sur une chaîne fonctionnelle $\mathcal{C} = {}^{v_0}\ F_1 \ldots F_n\ {}^{v_n}$, il faut garantir que les événements en entrée et en sortie dépendent l'un de l'autre, c'est-à-dire qu'il faut choisir $s_{v_0_S}[i]$ et $s_{v_n_A}[j]$ tels que $s_{v_0_S}[i] \quad s_{v_n_A}[j]$, mais il faut également garantir que $s_{v_n_A}[j]$ est le premier événement en sortie de la chaîne qui dépend de $s_{v_n_A}[j]$. Il est donc nécessaire d'ajouter une contrainte lors de la traversée d'une fonction pour garantir que l'exécution de la fonction est la première à traiter l'événement suivi. Pour la $i^{\text{ème}}$ occurrence de la variable x, délivrée au travers du canal temporisé C, et traitée lors de la $k^{\text{ième}}$ exécution de la fonction f, cela donne :

$$t_i^{x_C} \leq t_k^f \quad t_{k-1}^f < t_i^{x_C} \tag{7.28}$$

Le même principe s'applique lors de la constitution d'un système de contraintes pour une exigence de cohérence entre des chaînes fonctionnelles divergentes.

Dans la suite, nous présentons le prototype mettant en œuvre la construction du système de contraintes nécessaire à la vérification d'une exigence.

7.3 Mise en uvre et outillage

Nous avons réalisé un logiciel en *java* permettant la vérification d'exigences temps réel d'un système IMA. Nous avons implémenté les algorithmes mettant en œuvre l'approche présentée à la section pré-

cédente, ainsi que les algorithmes de l'*approche par trajectoire* qui permettent le calcul des temps de traversée du réseau. Deux librairies *open source* ont été utilisées : `RunCC` qui facilite le traitement de fichier en entrée de notre outil et `lp_solve` qui est un solveur de programmes linéaires.

L'implémentation des algorithmes de l'*approche par trajectoire* représente environ 600 lignes de code *java* et l'ensemble des autres traitements environ 3600 lignes de code. Cet outil a également été utilisé dans le cadre du projet SATRIMMAP. Une interface de 250 lignes de code a été développée pour accéder aux bases de données du projet.

7.3.1 Présentation générale

Nous rappelons les différentes étapes (c.f. section 3.2 page 53) de l'analyse d'un système :
- l'acquisition de l'architecture et des exigences du système,
- la création des objets représentant le système,
- la vérification de la correction du système vis-à-vis des propriétés définies au chapitre 4,
- le calcul des temps de traversée du réseau,
- pour chaque exigence, la création d'un sous-système abstrait constitué uniquement des processus impliqués dans l'exigence. Puis le parcours de ces processus pour la génération de l'ensemble des contraintes de l'exigence.

7.3.2 Acquisition et création des objets

L'entrée du prototype est un fichier texte décrivant l'architecture fonctionnelle, matérielle, l'allocation de la première sur la seconde, ainsi que l'ensemble des exigences temps réel que le système doit satisfaire. Cette description doit respecter une syntaxe que nous avons définie à l'aide du métalangage EBNF [57] (pour *Extented Backus-Naur Form*). L'intérêt d'utiliser le métalangage EBNF est qu'il existe de nombreuses librairies dédiées à l'analyse de fichiers exprimés sous cette forme. En particulier, nous avons utilisé la librairie *java open source* `RunCC` [49]. Nous donnons en exemple la syntaxe d'un RDC exprimée en EBNF, ainsi que les éléments nécessaires à sa définition, c'est-à-dire une période, un temps pire cas d'exécution (WCET), un port de communication et des éléments terminaux tels que les entiers naturels et les unités de temps et de débit :

```
// syntaxe d un RDC
RDC ::= "RDC"  identifier  periode WCET port "end" ";" ;

// syntaxe d une période
periode ::= "periode"  nat   uniteTps ";" ;

// syntaxe d un temps maximum d exécution
WCET ::= "WCET"  nat   uniteTps  ";" ;

// syntaxe d un port
port ::= "port"  identifier  debit "end" ";" ;

// syntaxe d un débit
debit ::= "debit"  nat   uniteDebit  ";" ;

// Terminaux
nat ::= ( 1 .. 9 )( 0 .. 9 )* |  0  ;
uniteTps ::= "ms" | "us" ;
uniteDebit ::= "b/s" | "kb/s" | "Mb/s" | "Gb/s" ;
```

Dans ce langage, la déclaration du RDC R_1 se fait alors de la façon suivante :

```
RDC R1
periode 50 ms ;
WCET 10 ms ;
port R1.1 debit 100 Mb/s ; ;
end ;
```

La syntaxe complète de ce langage est donnée dans l'annexe A. A partir de la définition de cette syntaxe, la librairie `runCC` génère un *parseur* dont la fonction est de vérifier si la syntaxe du fichier décrivant l'architecture à analyser est correcte. Le *parseur* fournit également un ensemble de méthodes permettant de parcourir cette description. A l'aide de ces méthodes, nous instancions l'ensemble des objets *java* représentant les éléments de l'architecture.

Dans le projet SATRIMMAP, l'ensemble des données relatives à un système est stocké dans une base de données. Une interface dédiée a donc été développée pour récupérer dans cette base les informations nécessaires à la définition d'une architecture. Dans ce cas de figure, le format d'entrée présenté précédemment n'est donc pas utilisé.

Correction du système. Une fois que l'ensemble des objets représentant le système a été créé, les propriétés de correction du système, définies au chapitre 4, sont testées.

7.3.3 Analyse réseau

Si le système est correct, nous procédons à l'analyse du réseau pour déterminer les meilleurs et pires temps de traversée le long de chaque chemin. A noter que le réseau est complément analysé, indépendamment des exigences à vérifier. Ces résultats permettent alors de paramétrer les objets représentant les canaux temporisés qui sont utilisés pour abstraire le réseau. Nous avons implémenté les algorithmes de l'*approche par trajectoire* pour cette analyse. Nous avons pu tester ces algorithmes sur la configuration d'un Airbus A380, constituée d'environ 1000 liens virtuels. L'analyse des 5000 chemins de ces liens virtuels prend entre 4 et 5 secondes.

7.3.4 Constitution de l'ensemble des contraintes relatives à une exigence

La constitution de l'ensemble des contraintes relatives à une exigence s'effectue en trois étapes. Dans un premier temps, nous instancions une représentation d'un sous-système abstrait. Concrètement, il s'agit d'instancier les objets qui représentent les processus intervenant dans l'exigence. Ces objets sont alors utilisés dans une deuxième étape. Leur rôle est de faciliter la création du graphe des événements que nous utiliserons pour évaluer l'exigence. Chaque processus implémente alors une méthode permettant de créer l'ensemble des événements qui dépendent d'un événement fourni en entrée. Par exemple, si nous donnons l'événement $s_{u_vl}[i]$ en entrée de l'objet du canal temporisé C, l'objet crée l'événement $s_{u_C}[i]$ tel que $s_{u_vl}[i] \quad s_{u_C}[i]$. A noter que les processus peuvent également être utilisés en remontant les dépendances : en fournissant un événement $s_{u_C}[j]$, l'objet crée l'événement $s_{u_vl}[j]$ dont il dépend.

Les objets représentant les capteurs et les actionneurs permettent également de générer un événement « spontanément », c'est-à-dire sans entrée particulière. L'événement ainsi généré sert de racine pour la création du graphe des événements. Par exemple, pour une exigence de latence, nous générons un tel événement sur le capteur en entrée de la chaîne fonctionnelle. Ensuite, nous utilisons les processus pour créer tous les événements dépendant de cet événement racine, jusqu'à l'actionneur en sortie de la chaîne. Nous procédons de la même façon pour une exigence de cohérence entre chaînes fonctionnelles divergentes. Pour une exigence de fraîcheur ou une exigence de cohérence entre chaînes fonctionnelles convergentes, nous procédons dans l'autre sens. Un événement est généré dans l'actionneur en sortie de la chaîne fonctionnelle et les dépendances sont alors remontées jusqu'au(x) capteur(s) en entrée de la chaîne. De la sorte, nous n'avons qu'un unique événement à générer « spontanément », les autres étant construits automatiquement par les processus.

Une fois que nous disposons de ce graphe d'événements, la troisième étape consiste en la création du système de contraintes qui a été décrit à la section 7.2. La librairie `lp_solve` est alors utilisée. Nous en utilisons les différentes interfaces de programmation (API pour *Application Programming Interface*) pour déclarer un programme linéaire mixte avec toutes les contraintes existant entre les événements du graphe, créé lors de la deuxième étape, ainsi que sa fonction objectif. Il est alors possible de demander la résolution de ce programme pour vérifier l'exigence.

Chapitre 8

Résolution

Dans le chapitre précédent, nous avons décrit la constitution d'un système de contraintes permettant d'écrire le comportement temporel de la (ou les) chaîne(s) fonctionnelle(s) d'une exigence. A présent, nous proposons d'exploiter ce système de contraintes pour vérifier la satisfaction de l'exigence. Cette méthode (section 8.1) repose sur l'utilisation d'un solveur de programme linéaire mixte. Le pire ou meilleur cas d'une propriété, exprimé dans le *tagged signal model* au chapitre 6, est alors exprimé sous la forme d'une fonction objectif (section 8.2) d'un MILP (pour *Mixed Integer Linear Program*) constitué du système de contraintes. La résolution de ce MILP permet alors d'obtenir la valeur exacte du pire ou meilleur cas. La méthode est dite exacte parce qu'elle explore l'ensemble des comportements du système étudié. Les résultats obtenus sur l'étude de cas (section 8.3) sont comparés aux résultats fournis par des formules *locales*, par opposition à notre méthode proposant une démarche *globale*, donnant un majorant du pire ou meilleur cas. Les approches utilisant la programmation linéaire mixte peuvent être coûteuses en terme de temps de calcul, ainsi nous testons la robustesse de notre méthode vis-à-vis du passage à l'échelle au travers d'expérimentations (chapitre 9). Les observations faites grâce à notre méthode nous permettrons également de proposer des améliorations pour les formules locales.

8.1 Encodage sous la forme d'un programme linéaire mixte

Une fois les contraintes relatives à une ou plusieurs chaînes fonctionnelles déterminées, il est envisageable d'utiliser un solveur pour rechercher le pire ou meilleur cas d'une exigence. Il est alors nécessaire de préciser au solveur le domaine de définition des variables du système de contraintes. En outre, certaines contraintes apparaissant dans le système nécessitent une ré-écriture pour être utilisées dans un solveur. Finalement, nous introduisons la notion de fonction objectif du problème. La section 8.2 sera consacrée à la définition des fonctions objectifs par type d'exigence.

8.1.1 Domaine de définition des variables

Le système de contraintes constitué pour une exigence est composé d'un ensemble de variables discrètes et continues. Toutes les variable discrètes sont par défaut dans $\mathbb{N}$ et toutes les variables continues sont par défaut dans $\mathbb{R}^+$. Ces variables peuvent alors être contraintes pour ne pas dépasser certaines valeurs.

Pour les variables discrètes :

- les indices d'événements n'ont pas de limites particulières,
- le nombre de *bag* c subi dans un lisseur de trafic vl est limité par : $c \leq N_{vl} - 1$.

Pour les variables continues :

- le temps d'exécution δ_f d'une fonction f est limité par : $\delta_f \leq C_f$,
- le temps de traversée δ_C d'un canal temporisé C est limité par : $a_C \leq \delta_C$ et $\delta_C \leq b_C$,
- la phase ϕ_M d'un module M est limitée par : $\phi_M \leq$ *HP(M)*, avec *HP(M)* l'hyperpériode du module,
- la phase ϕ_S d'un capteur S est limitée par $\phi_S \leq T_S$, avec T_S la période du capteur,
- les estampilles sont contraintes par une valeur minimale que nous détaillons dans la suite.

En effet, les estampilles des événements ne doivent pas être choisies complètement arbitrairement, car nous cherchons un ensemble d'événements représentatif du système en fonctionnement et non pas représentatif de sa phase d'initialisation. La figure 8.1 donne deux mesures de la fraîcheur pire cas sur un exemple élémentaire : un capteur S produit une variable consommée par une fonction f qui s'exécute

avec la fonction g sur le module M. Le chronogramme du haut représente le pire cas de fraîcheur F qui est mesuré au niveau de l'instant initial. Le chronogramme du bas illustre le pire cas d'un autre comportement (ϕ_S et ϕ_M sont donc différents du comportement précédent) si l'on ne considère pas le cas particulier de l'initialisation. On peut constater que ces deux fraîcheurs ne sont pas égales. Seule la fraîcheur pire cas en fonctionnement est pertinente pour notre analyse.

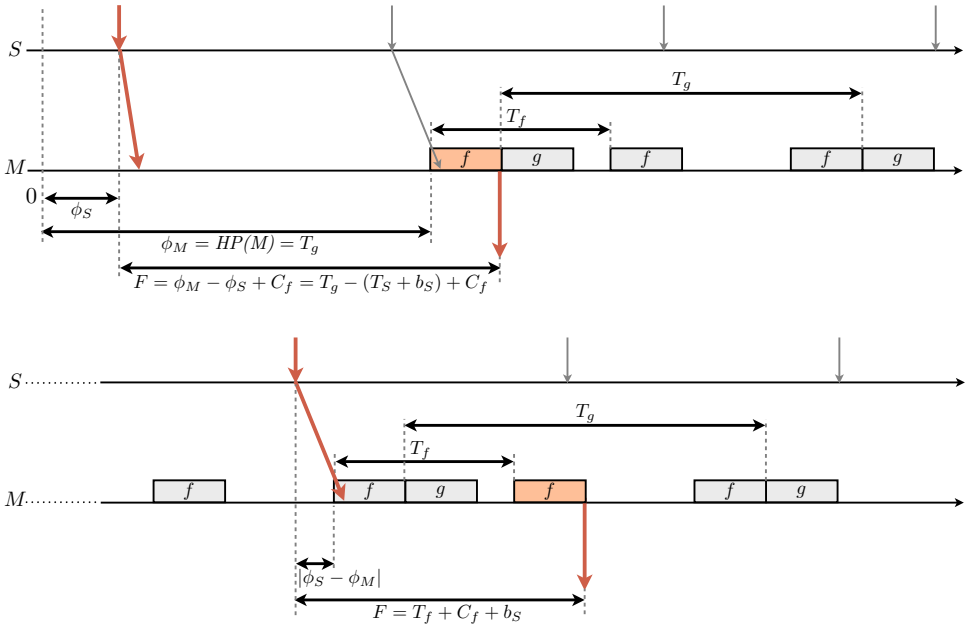


FIGURE 8.1 – Fraîcheur pire cas à l'initialisation et en fonctionnement

Pour éviter que le résultat de l'analyse ne considère le cas particulier de l'instant initial, on ajoute des contraintes sur les estampilles des événements considérés. Les événements considérés en entrée des chaînes fonctionnelles doivent survenir après l'initialisation de tous les modules et tous les capteurs. Dans un système de contraintes, tel que défini au chapitre précédent et dédié à l'analyse d'une exigence φ, toute estampille t_i^{u-S} en entrée d'une chaîne fonctionnelle doit respecter :

$$t_i^{u-S} \geq \max \left\{ \max_{m \ \ Modules(\varphi)} HP(m) \ ; \ \max_{s \ \ Sensors(\varphi)} T_s \right\} \tag{8.1}$$

avec :

- $Modules(\varphi)$ l'ensemble des modules intervenant dans l'exigence φ et $HP(m)$ l'hyper-période du module m,
- $Sensors(\varphi)$ l'ensemble des capteurs intervenant dans l'exigence φ et T_s la période du capteur s,

Le maximum des hyper-périodes des modules et des périodes des capteurs n'étant pas une variable du système de contraintes, il est utilisé comme un paramètre du problème, c'est-à-dire qu'il est calculé en amont de la résolution.

8.1.2 Particularité pour la cohérence

Le système de contraintes constitué dans le cadre d'une exigence de cohérence nécessite quelques aménagements pour son utilisation avec un solveur. Ces particularités concernent les contraintes sur les temps de traitement d'une fonction et les lisseurs de trafic. Dans la suite, nous considérons deux chaînes fonctionnelles se croisant au niveau d'une fonction f, telles que représentées sur la figure 8.2, avec les dépendances des variables (u, v) et $(u\ , v\ )$. Ces chaînes sont impliquées dans une même exigence de cohérence (divergente ou convergente).

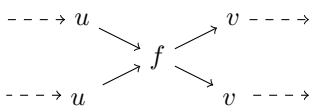


FIGURE 8.2 – Croisement de chaînes fonctionnelles au niveau de f

Temps de traitement de la fonction

Lors de la construction d'un système de contraintes pour ces chaînes fonctionnelles, les deux contraintes suivantes, issues du processus P_f de la fonction f, apparaissent :

$$\begin{array}{ll} \delta & [0, C_f] \text{ t.q. } t_i^{u_f} = t_k^f + \delta \\ \delta & [0, C_f] \text{ t.q. } t_{i'}^{u'_f} = t_{k'}^f + \delta \end{array} \tag{8.2}$$

ainsi qu'une contrainte supplémentaire garantissant que lors d'une exécution, toutes les sorties d'une fonction sont produites à une même date :

$$\text{si } k = k \ \text{ alors } \delta = \delta \tag{8.3}$$

Cependant, cette dernière contrainte, conditionnelle, n'est pas directement utilisable dans un solveur de programmes linéaires. L'implémentation de cette contrainte se fait alors à l'aide des contraintes suivantes :

$$\begin{cases} k \leq k \ + b_1 \cdot M & (1) \\ k \ \leq k + b_1 \cdot M & (2) \\ \delta \leq \delta \ + b_1 \cdot M & (3) \\ \delta \ \leq \delta + b_1 \cdot M & (4) \\ k < k \ + b_2 \cdot M + (1 - b_1) \cdot M & (5) \\ k \ < k + (1 - b_2) \cdot M + (1 - b_1) \cdot M & (6) \end{cases} \tag{8.4}$$

Deux variables booléennes b_1 et b_2 sont utilisées ainsi que M un grand entier représentant l'infini. Le choix de M sera discuté ultérieurement (c.f. sous-section 8.1.3). Si le solveur choisit $b_1 = 0$, alors les contraintes (1) à (4) forcent $k = k$ et $\delta = \delta$, et les contraintes (5) et (6) sont alors inhibées, c'est-à-dire qu'elles sont toujours vraies quels que soient k, k et b_2. En revanche, si le solveur choisit $b_1 = 1$, cela revient à inhiber les contraintes (1) à (4) et à activer les contraintes (5) et (6). Dans ce cas, les valeurs de δ et δ ne sont plus contraintes l'une par rapport à l'autre et, nécessairement, $k = k$. En effet, suivant la valeur de b_2, on obtient soit $k < k$, soit $k \ < k$. La contrainte (8.3) est ainsi bien implémentée.

Lisseur de tra c

Considérons également que les variables v et v sont transmises au travers d'un même lisseur de trafic vl en sortie de f. Alors, les contraintes suivantes, issues du processus abstrait A_{vl} (c.f. définition (7.5) page 112), font partie du système de contraintes :

$$\begin{cases} c \quad 0..(N_{vl} - 1) \text{ t.q. } t_i^{u_vl} = t_i^{u_f} + c \cdot bag \\ c \quad 0..(N_{vl} - 1) \text{ t.q. } t_{i'}^{u'_vl} = t_i^{u'_f} + c \ \cdot bag \\ t_i^{u_vl} = t_{i'}^{u'_vl} \end{cases} \tag{8.5}$$

La contrainte $t_i^{u_vl} = t_{i'}^{u'_vl}$ n'est pas directement utilisable par un solveur. Nous la remplaçons donc par les contraintes suivantes :

$$\begin{cases} t_i^{u_vl} & < t_{i'}^{u'_vl} + b_{vl} \cdot M \\ t_{i'}^{u'_vl} & < t_i^{u_vl} + (1 - b_{vl}) \cdot M \end{cases} \tag{8.6}$$

avec b_{vl} une variable booléenne et M un grand entier représentant l'infini. Avec ces contraintes, pour toutes valeurs de b_{vl}, on a nécessairement $t_i^{u_vl} = t_{i'}^{u'_vl}$.

8.1.3 Représentation de l'infini

Nous avons vu dans la sous-section précédente que nous utilisons un grand entier M pour inhiber certaines contraintes. Par exemple, la contrainte $x \leq y + b \cdot M$ est inhibée si le booléen $b = 1$ et elle est activée sinon. Il pourrait être tentant de choisir pour valeur de M un entier tel que 10^{30}, mais comme précisé dans la documentation du solveur `lp_solve`, ce choix pourrait amener à des résultats erronés. En effet, lors de la résolution du programme, la valeur utilisée pour un booléen b à zéro peut ne pas être 0, mais plutôt une valeur très faible comme 10^{-20} par exemple. Multiplié par M, le résultat n'est plus zéro mais 10^{10}, ce qui change le sens de la contrainte. En pratique, il faut donc choisir un entier « réaliste », c'est-à-dire grand par rapport à la ou les contraintes contrôlées à l'aide de M. Prenons par exemple la contrainte $t_i^{u_vl} < t_{i'}^{u'_vl} + b_{vl} \cdot M$ issue du processus d'un lisseur de trafic (c.f. équation (8.6)). Les estampilles $t_i^{u_vl}$ et $t_{i'}^{u'_vl}$ représentent les instants de sortie d'occurrences de variables qui ont été produites lors d'une même exécution. Ainsi, quelle que soit la valeur des estampilles, elles ne peuvent être distantes de plus de $(N_{vl} - 1) \cdot bag$, N_{vl} étant le nombre maximal d'occurrences traitées par le lisseur de trafic lors d'une exécution. Si M est strictement plus grand que $N_{vl} \cdot bag$, alors il est possible d'inhiber ou d'activer la contrainte avec $b_{vl} \cdot M$. Pour l'étude de cas, nous utilisons, pour l'infini, la valeur sûre $M = 10^7$.

8.1.4 Fonction objectif

La fonction objectif sert à préciser au solveur son objectif d'optimisation. Pour un programme linéaire mixte, elle s'exprime sous la forme suivante :

$$\text{min ou max} : a_1 \cdot x_1 + a_2 \cdot x_2 + \ldots + a_n \cdot x_n \tag{8.7}$$

avec :
- $a_1, \ldots, a_n$ $\mathbb{R}^n$ un ensemble de coefficients,
- $x_1, \ldots, x_n$ un ensemble de variables du problème continues ou discrètes,
- max un objectif de maximisation,
- min un objectif de minimisation.

Dans le section suivante, nous donnons les fonctions objectifs exprimant les pires et meilleurs cas des exigences latence, fraîcheur et cohérence.

8.2 Fonctions objectifs par type d'exigence

Pour chaque exigence temps réel, nous définissons une fonction objectif permettant d'obtenir le pire cas de l'exigence et, si il y a lieu, le meilleur cas.

8.2.1 Exigence de latence

Latence pire cas

La définition de la latence pire cas d'une chaîne fonctionnelle $\mathcal{C} = {}^{v_0}\ F_1 \ldots F_n\ {}^{v_n}$ (c.f. chapitre 6 page 96) est :

$$WCL(\mathcal{C}) = \max_{\mathbf{s}\ \mathbb{S}} \left\{ \max_{\substack{(i,j) \in \mathbb{N}^2 \\ s_{v_0_S}[i] \rightarrow s_{v_n_A}[j] \\ s_{v_0_S}[i] \nrightarrow s_{v_n_A}[j-1]}} t_j^{v_n_A} - t_i^{v_0_S} \right\} \tag{8.8}$$

Considérons que lors de la constitution du système de contraintes associé à cette chaîne fonctionnelle, l'estampille de l'événement en entrée est $t_i^{v_0_S}$ et l'estampille de l'événement en sortie est $t_j^{v_n_A}$. Le système de contraintes reliant l'événement $s_{v_0_S}[i]$ à l'événement $s_{v_n_A}[j]$ garantit que $s_{v_n_A}[j]$ est le premier événement dépendant de $s_{v_0_S}[i]$ en sortie de la chaîne. Ainsi, la fonction objectif s'exprime simplement :

$$\max : t_j^{v_n_A} - t_i^{v_0_S} \tag{8.9}$$

Latence meilleure cas

De manière similaire à la latence pire cas, la latence meilleure cas de la chaîne $\mathcal{C} = {}^{v_0}\ F_1 \dots F_n\ {}^{v_n}$ s'obtient avec la fonction objectif suivante :

$$\min : t_j^{v_n_A} - t_i^{v_0_S} \tag{8.10}$$

8.2.2 Exigence de fraîcheur

Comme précisé lors de la définition de la fraîcheur (c.f. sous-section 6.2.2 page 97), seule la notion de fraîcheur pire cas est pertinente car la notion de fraîcheur meilleur cas est équivalente à la latence meilleur cas. La définition de la fraîcheur pire cas pour une chaîne fonctionnelle $\mathcal{C} = {}^{v_0}\ F_1 \dots F_n\ {}^{v_n}$ est :

$$WCF(\mathcal{C}) = \max_{\mathbf{s}\ \mathbb{S}} \left\{ \max_{\substack{(i,j) \in \mathbb{N}^2 \\ s_{v_0_S}[i] \to s_{v_n_A}[j]}} t_j^{v_n_A} - t_i^{v_0_S} \right\} \tag{8.11}$$

La fonction objectif s'exprime directement :

$$\max : t_j^{v_n_A} - t_i^{v_0_S} \tag{8.12}$$

8.2.3 Exigence de cohérence entre chaînes fonctionnelles divergentes

Pour les exigences de cohérence, nous exprimons la fonction objectif pour une exigence avec plus de deux chaînes fonctionnelles. Soit un ensemble de chaînes fonctionnelles divergentes $\mathcal{C}_1, \dots, \mathcal{C}_n$ tel que pour tout $i = 1..n$, la chaîne $\mathcal{C}_i$ se termine par la fonction F_i produisant la variable v_i traitée par l'actionneur A_i. Toutes ces chaînes commencent par la variable u, produite par le capteur S.

Pire cas de cohérence entre chaînes fonctionnelles divergentes

L'expression du pire cas de cohérence est donnée au chapitre 6 (formule (6.20) page 100) pour deux chaînes fonctionnelles. Etendues à n chaînes, cette expression devient :

$$WCCD(\mathcal{C}_1, \dots, \mathcal{C}_n) = \max_{\mathbf{s}\ \mathbb{S}} \left\{ \max_{\substack{(i, j_1, \dots, j_n) \in \mathbb{N}^{n+1}, \\ \forall k = 1..n, s_{u_S}[i] \to s_{v_k_A_k}[j_k], \\ j_k = \min\{l \in \mathbb{N} \mid s_{u_S}[i] \to s_{v_k_A_k}[l]\}}} \max_{k=1..n} t_{j_k}^{v_k_A_k} - \min_{k=1..n} t_{j_k}^{v_k_A_k} \right\} \tag{8.13}$$

Considérons que lors de la constitution du système de contraintes associées à ces chaînes fonctionnelles, l'ensemble des estampilles temporelles des événements en sortie des chaînes est $t_{j_1}^{v_1_A_1}, \dots, t_{j_n}^{v_n_A_n}$. Le pire cas de cohérence sur ces chaînes s'exprime comme le maximum de la différence entre la plus grande et la plus petite des estampilles, soit :

$$\max : \max_{i=1..n} t_{j_i}^{v_i_A_i} - \min_{i=1..n} t_{j_i}^{v_i_A_i} \tag{8.14}$$

Mais pour être utilisée dans un solveur, cette expression doit être linéarisée. Nous introduisons deux variables intermédiaires représentant le minimum et le maximum des estampilles des événements en sortie des chaînes, soit $minT$ et $maxT$ respectivement. La fonction objectif devient alors :

$$\max : maxT - minT \tag{8.15}$$

Nous ajoutons certaines contraintes pour garantir qu'au moins une des estampilles est égale à $maxT$ et au moins une à $minT$. Comme l'objectif est de maximiser la différence $maxT - minT$, le solveur choisira ces variables de telle sorte que $maxT$ soit égale à la plus grande estampille et que $minT$ soit égale à la plus petite.

Nous commençons par forcer $maxT$ à être égale à au moins une des estampilles. Pour chaque estampille $t_{j_i}^{v_i - A_i}$, nous ajoutons les deux contraintes suivantes :

$$\begin{aligned} maxT &\leq t_{j_i}^{v_i - A_i} + b_i^{maxT} \cdot M \\ maxT &\geq t_{j_i}^{v_i - A_i} - b_i^{maxT} \cdot M \end{aligned} \tag{8.16}$$

où b_i^{maxT} est une variable booléenne et M est un grand entier représentant l'infini. Avec ces contraintes, si $b_i^{maxT} = 0$ alors nécessairement $t_{j_i}^{v_i - A_i} = maxT$ et sinon aucune contrainte n'est imposée entre $maxT$ et $t_{j_i}^{v_i - A_i}$. On ajoute alors la contrainte suivante pour garantir qu'au moins une des estampilles soit égale à $maxT$, c'est-à-dire qu'une des variables booléennes doit être égale à 0 et les autres doivent être égales à 1 :

$$\sum_{i=1}^{n} b_i^{maxT} = n - 1 \tag{8.17}$$

On procède de la même façon pour garantir qu'au moins une des estampilles soit égale à $minT$. Pour tout $i = 1..n$, on ajoute les contraintes :

$$\begin{aligned} minT &\leq t_{j_i}^{v_i - A_i} + b_i^{minT} \cdot M \\ minT &\geq t_{j_i}^{v_i - A_i} - b_i^{minT} \cdot M \end{aligned} \tag{8.18}$$

Comme pour le maximum, la contrainte suivante doit être ajoutée :

$$\sum_{i=1}^{n} b_i^{minT} = n - 1 \tag{8.19}$$

L'expression de la fonction objectif pour le calcul du pire cas induit l'ajout dans le système de contraintes de 2 variables continues, $2n$ variables booléennes et $4n + 2$ contraintes, où n est le nombre de chaînes fonctionnelles impliquées dans l'exigence. Dans l'annexe B, nous donnons en exemple la fonction objectif, ainsi que les contraintes qui y sont associées, de l'exigence de cohérence entre chaînes divergentes de l'étude de cas.

Meilleur cas de cohérence entre chaînes fonctionnelles divergentes

Le meilleur cas de cohérence sur ces chaînes s'exprime comme le minimum de la différence entre la plus grande et la plus petite des estampilles, soit :

$$\min : \max_{i=1..n} t_{j_i}^{v_i - A_i} - \min_{i=1..n} t_{j_i}^{v_i - A_i} \tag{8.20}$$

La notion de meilleur cas de cohérence étant moins intuitive que la notion de pire cas, nous l'illustrons sur la figure 8.3. Nous représentons les plages de valeurs des estampilles des événements $s_{v_1_A_1}[j_1]$, $s_{v_2_A_2}[j_2]$ et $s_{v_3_A_3}[j_3]$ en sortie des chaînes $\mathcal{C}_1$, $\mathcal{C}_2$ et $\mathcal{C}_3$. Ces plages de valeurs dépendent des contraintes du programme linéaire. Le meilleur cas de cohérence est obtenu en minimisant la distance entre la plus grande estampille et la plus petite. Sur cet exemple, cela correspond à assigner la plus petite valeur possible à $t_{j_2}^{v_2 - A_2}$, qui correspond alors à la plus grande estampille, et la plus grande valeur possible à $t_{j_3}^{v_3 - A_3}$, qui correspond alors à la plus petite estampille, finalement, $t_{j_1}^{v_1 - A_1}$ prend une valeur intermédiaire.

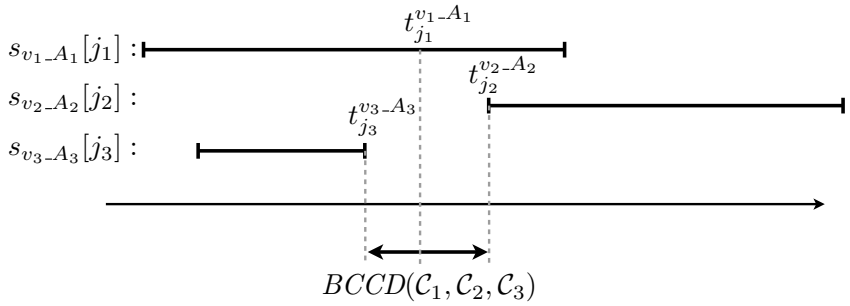


FIGURE 8.3 – Illustration du meilleur cas de cohérence

Comme pour le pire cas, les variables intermédiaires $maxT$ et $minT$ sont introduites pour linéariser la fonction objectif. Elle devient alors :

$$\min : maxT - minT \tag{8.21}$$

Les mêmes contraintes sont alors ajoutées pour garantir que $maxT$ et $minT$ correspondent à des valeurs des estampilles. Dans la recherche du pire cas, comme l'objectif est une maximisation, cela garantit que $maxT$ est le maximum des estampilles et $minT$ est le minimum. En revanche, pour le meilleur cas, nous devons ajouter des contraintes supplémentaires, sinon le processus de minimisation attribuerait la plus petite estampille à $maxT$ et la plus grande à $minT$. Pour chaque estampille $t_{j_i}^{v_i_A_i}$, nous ajoutons les deux contraintes suivantes :

$$\begin{aligned} maxT &\geq t_{j_i}^{v_i_A_i} \\ minT &\leq t_{j_i}^{v_i_A_i} \end{aligned} \tag{8.22}$$

Ainsi, la variable $maxT$ est contrainte à être égale à la plus grande estampille et $minT$ est contrainte à être égale à la plus petite estampille. L'expression de la fonction objectif pour le calcul du meilleur cas de cohérence induit l'ajout dans le système de contraintes de 2 variables continues, $2n$ variables booléennes et de $6n + 2$ contraintes, avec n le nombre de chaînes fonctionnelles impliquées dans l'exigence.

8.2.4 Exigence de cohérence entre chaînes fonctionnelles convergentes

Nous exprimons la fonction objectif pour une exigence de cohérence entre n chaînes fonctionnelles convergentes. Soit un ensemble de chaînes fonctionnelles convergentes $\mathcal{C}_1, \ldots, \mathcal{C}_n$ tel que pour tout $i = 1..n$, la chaîne $\mathcal{C}_i$ commence par la variable u_i produite par le capteur S_i. Toutes ces chaînes se terminent par la variable v produite par l'actionneur A.

Pire cas de cohérence entre chaînes fonctionnelles convergentes

L'expression du pire cas de cohérence est donnée au chapitre 6 (formule (6.23) page 102) pour deux chaînes fonctionnelles. Etendue à n chaînes, cette expression devient :

$$WCCC(\mathcal{C}_1, \ldots, \mathcal{C}_n) = \max_{\mathbf{s}\ \mathbb{S}} \left\{ \max_{\substack{(i_1, \ldots, i_n, j) \in \mathbb{N}^{n+1}, \\ \forall k = 1..n,\, s_{u_k_S_k}[i_k] \rightarrow s_{v_A}[j]}} \max_{k=1..n} t_{i_k}^{u_k_S_k} - \min_{k=1..n} t_{i_k}^{u_k_S_k} \right\} \tag{8.23}$$

Considérons que lors de la constitution du système de contraintes associées à ces chaînes fonctionnelles, l'ensemble des estampilles temporelles des événements en entrée des chaînes est $t_{i_1}^{u_1_S_1}, \ldots, t_{i_n}^{u_n_S_n}$. Le

pire cas de cohérence sur ces chaînes s'exprime comme le maximum de la différence entre la plus grande et la plus petite des estampilles, soit :

$$\max : \max_{k=1..n} t_{i_k}^{u_k - S_k} - \min_{k=1..n} t_{i_k}^{u_k - S_k} \tag{8.24}$$

Comme pour l'exigence de cohérence entre chaînes divergentes, il est nécessaire de linéariser cette expression. Nous utilisons une méthode similaire, c'est-à-dire que nous introduisons des variables intermédiaires $maxT$ et $minT$, qui représentent respectivement la plus grande et la plus petite estampille. La fonction objectif devient alors :

$$\max : maxT - minT \tag{8.25}$$

Avec les contraintes supplémentaires, pour toute estampille $t_{i_k}^{u_k - S_k}$:

$$\begin{aligned} maxT &\leq t_{i_k}^{u_j - S_k} + b_k^{maxT} \cdot M \\ maxT &\geq t_{i_k}^{u_k - S_k} - b_k^{maxT} \cdot M \\ minT &\leq t_{i_k}^{u_k - S_k} + b_k^{minT} \cdot M \\ minT &\geq t_{i_k}^{u_k - S_k} - b_k^{minT} \cdot M \end{aligned} \tag{8.26}$$

Ainsi que les deux contraintes sur les variables booléennes b_k^{maxT} et b_k^{minT} :

$$\sum_{k=1}^{n} b_k^{maxT} = n - 1 \quad \sum_{k=1}^{n} b_k^{minT} = n - 1 \tag{8.27}$$

L'expression de la fonction objectif pour le calcul du pire cas induit l'ajout dans le système de contraintes de 2 variables continues, $2n$ variables booléennes et $4n + 2$ contraintes, avec n le nombre de chaînes fonctionnelles impliquées dans l'exigence.

Meilleur cas de cohérence entre chaînes fonctionnelles convergentes

Le meilleur cas de cohérence sur ces chaînes s'exprime comme le minimum de la différence entre la plus grande et la plus petite des estampilles, soit :

$$\min : \max_{k=1..n} t_{i_k}^{u_k - S_k} - \min_{k=1..n} t_{i_k}^{u_k - S_k} \tag{8.28}$$

Comme pour le pire cas, les variables intermédiaires $maxT$ et $minT$ sont introduites pour linéariser cette fonction objectif qui devient alors :

$$\min : maxT - minT \tag{8.29}$$

Comme pour les chaînes divergentes, il est également nécessaire d'ajouter, pour chaque estampille $t_{i_k}^{u_k - S_k}$, les contraintes :

$$\begin{aligned} maxT &\leq t_{i_k}^{u_k - S_k} + b_k^{maxT} \cdot M \\ maxT &\geq t_{i_k}^{u_k - S_k} - b_k^{maxT} \cdot M \\ minT &\leq t_{i_k}^{u_k - S_k} + b_k^{minT} \cdot M \\ minT &\geq t_{i_k}^{u_k - S_k} - b_k^{minT} \cdot M \\ maxT &\geq t_{i_k}^{u_k - S_k} \\ minT &\leq t_{i_k}^{u_k - S_k} \end{aligned} \tag{8.30}$$

Ainsi que les deux contraintes sur les variables booléennes b_k^{maxT} et b_k^{minT} :

$$\sum_{k=1}^{n} b_k^{maxT} = n - 1 \quad \sum_{k=1}^{n} b_k^{minT} = n - 1 \tag{8.31}$$

L'expression de la fonction objectif pour le calcul du meillleur cas induit l'ajout dans le système de contraintes de 2 variables continues, $2n$ variables booléennes et $6n + 2$ contraintes, où n est le nombre de chaînes fonctionnelles impliquées dans l'exigence.

En conclusion, toutes les exigences définies au chapitre 6 ont pu être exprimées sous une forme permettant leur utilisation dans un MILP. La linéarisation d'une exigence de cohérence nécessite l'ajout de variables booléennes dans le programme linéaire. Nous observerons l'impact de cette linéarisation sur le temps de résolution du programme, au travers d'expérimentations présentées à la section 9.1.

8.3 Application à l'étude de cas

Dans cette section, nous appliquons notre méthode de vérification aux exigences de l'étude de cas qui ont été présentées au chapitre 6 à la page 102. Les programmes linéaires mixtes utilisés pour l'évaluation des exigences sont détaillés dans l'annexe B.

Pour évaluer le gain apporté par notre méthode, nous nous comparons aux résultats proposés dans [6]. Il s'agit de la formule la plus adaptée à l'évaluation d'une latence pire cas dans notre contexte. Nous proposons dans l'annexe C d'étendre ce résultat pour la latence meilleure cas, ainsi que pour les exigences de fraîcheur et de cohérence. Nous disposons ainsi de formules donnant une borne supérieure ou inférieure aux exigences pire ou meilleur cas. Dans la suite, nous qualifions ces formules de *locales*, dans le sens où un pire (ou un meilleur) scénario d'exécution est construit en considérant que chaque composant, le long d'une chaîne fonctionnelle, subit *localement* son pire cas d'exécution. Ceci peut mener à considérer des scénarios impossibles au niveau du système global, et rend ces formules potentiellement sur-approximatives. A l'opposé, notre méthode peut être qualifiée de *globale*, dans le sens où chaque scénario est, par nature, considéré au niveau global du système. Ainsi, dans la suite, nous comparons les résultats obtenus avec notre méthode *globale*, avec les résultats de la méthode *locale*. Au travers de ces expérimentations, nous observerons également la sensibilité des exigences à certains paramètres du système, notamment les délais de communication.

Un des avantages de la méthode proposée est de fournir en sortie un comportement réalisant le pire ou meilleur cas d'une exigence. En effet, en sortie du solveur, on obtient non seulement le maximum ou le minimum de la fonction objectif, mais également la valeur des variables du programme menant à cet optimum. On récupère ainsi les indices des événements du comportement, les estampilles des événements, ainsi que les phases des modules, les temps d'exécution...

8.3.1 Exigence de latence

Pour l'exigence de latence $Latency_{\leq 700}(\mathcal{L}_1)$, nous obtenons une latence pire cas $WCL(\mathcal{L}_1) = 450,4$ms. Ainsi, cette exigence de latence est satisfaite. Le comportement réalisant ce pire cas est représenté sur la figure 8.4. Les occurrences mises en valeur correspondent à celles utilisées le long de la chaîne fonctionnelle. Nous décrivons le déroulement de ce comportement :

- la saisie du pilote ($req_1[1]$), considérée en entrée de la chaîne, est produite à 199,8 ms,
- $req_1[1]$ est délivrée au module M_1 juste après l'activation de la fonction KU_1 avec un retard, dû au bus du capteur key_1, de 0.2 ms (la phase du module est nulle : $\phi_{M_1} = 0$ ms),
- l'occurrence suivante de KU_1 produit l'identifiant du *waypoint* ($wpId_1[1]$) à 275 ms.
- $wpId_1[1]$ est délivrée au module M_3 après avoir subi un retard maximal dans le canal temporisé, soit 0,394 ms,
- l'occurrence de FM_1 débutant à 326,54 ms traite $wpId_1[1]$ et produit $query_1[1]$ à 356,54 ms (la phase du module est $\phi_{M_3} = 26,54$ ms),
- $query_1[1]$ est délivrée au module M_7 après avoir subi un retard maximal dans le canal temporisé, soit 0,46 ms,
- l'occurrence de *NDB* débutant à 422,072 ms traite $query_1[1]$ et produit $answer_1[1]$ à 442,072 ms (la phase du module est $\phi_{M_7} = 22,072$ ms),
- $answer_1[1]$ est délivrée au module M_3 après avoir subi un retard maximal dans le lisseur de trafic, soit 64 ms et un retard maximal dans le canal temporisé, soit 0,46 ms,

– $answer_1[1]$ arrive juste après une activation de FM_1 et est par conséquent traitée par l'occurrence de la fonction débutant à 566,54 ms. Ce traitement est alors instantané : $wpInfo_1[1]$ est produite à 566,54 ms,
– $wpInfo_1[1]$ est délivrée au module M_1 après avoir subi un retard maximal dans le lisseur de trafic, soit 8 ms, et un retard maximal dans le canal temporisé, soit 0,46 ms, $wpInfo_1[1]$ arrive juste après une activation de MFD_1 et est par conséquent traitée par l'occurrence de la fonction débutant à 625 ms. Ce traitement dure un temps maximal : $disp_1[13]$ est produite à 650,2 ms. Il s'agit ici de la 13ième occurrence de la variable $disp_1$, car $disp_1$ est une variable périodique qui est produite systématiquement par MFD_1, indépendamment des arrivées de $wpInfo_1$.

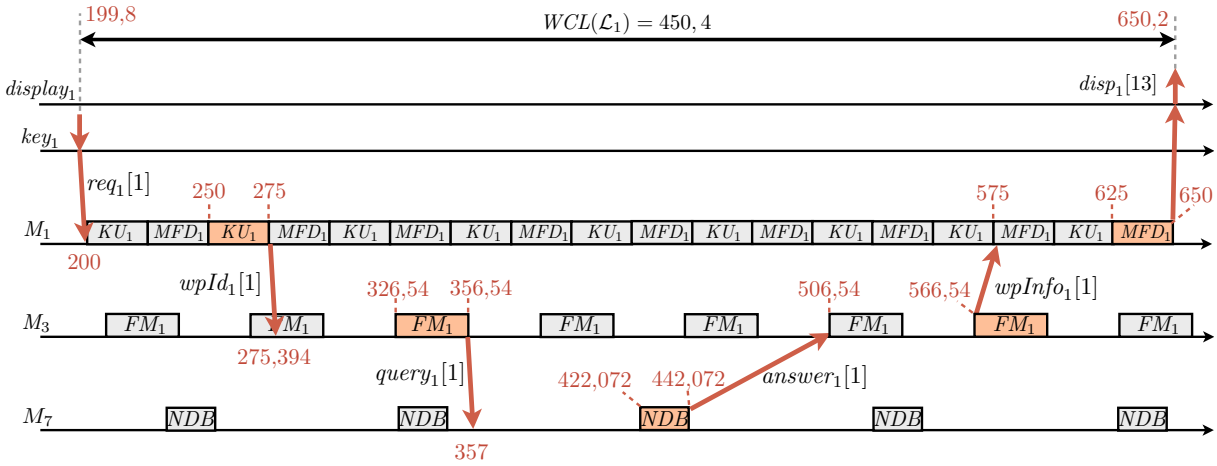


FIGURE 8.4 – Exemple d'un comportement donnant la latence pire cas de $\mathcal{L}_1$

Nous avons également évalué la latence meilleur cas pour la chaîne $\mathcal{L}_1$, ce qui donne $BCL(\mathcal{L}_1) = 75,2$ ms. Le comportement réalisant ce meilleur cas est représenté sur la figure 8.5.

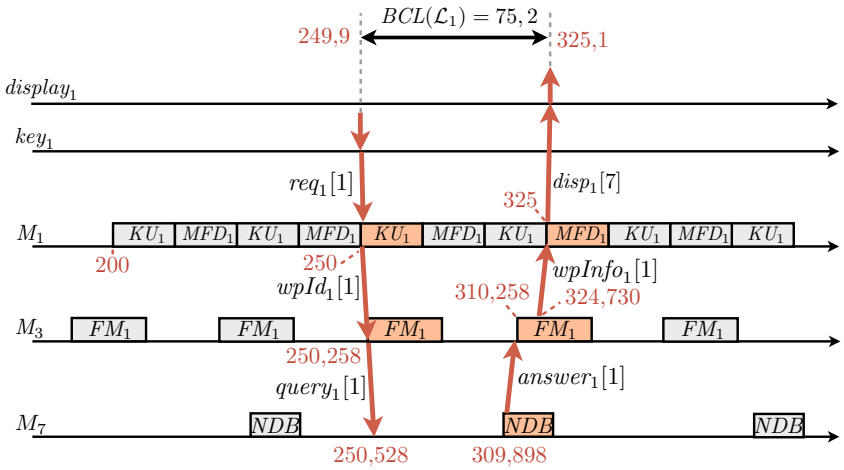


FIGURE 8.5 – Exemple d'un comportement donnant la latence meilleur cas $\mathcal{L}_1$

On peut remarquer sur ces figures que les comportements donnant les latences pire et meilleur cas ne sont pas uniques. Par exemple, sur la figure 8.4, c'est la même occurrence de MFD_1 qui traite $wpInfo_1[1]$ et ce, quel que soit le temps d'exécution de l'occurrence de FM_1 produisant $wpInfo_1[1]$. La latence pire cas est alors identique. De même, si le temps de traversée du réseau de $wpId_1[1]$ est plus important, dans une certaine mesure, $wpId_1[1]$ est traitée par la même occurrence de FM_1 et la latence pire cas est inchangée.

Ce qui nous amène à considérer le point suivant : quel est l'impact des délais de communication sur la latence ? Nous avons donc réalisé une série d'expériences dans le but d'observer la sensibilité de la latence aux délais de communication. Dans notre cas, un délai de communication comprend à la fois le temps

d'attente dans un lisseur de trafic et le temps de traversée d'un canal temporisé. Le programme linéaire permettant l'évaluation de la latence pire cas est alors modifié : nous forçons les variables représentant le nombre de *bag* subis dans un lisseur de trafic à zéro et nous faisons varier les bornes supérieures et inférieures des canaux temporisés. Pour simplifier la démarche, nous attribuons les mêmes bornes à tous les canaux. De cette façon, tous les délais de communication sont compris dans un intervalle $[\delta^{min}, \delta^{max}]$. Nous réalisons alors plusieurs évaluations de $WCL(\mathcal{L}_1)$ et $BCL(\mathcal{L}_1)$ en faisant varier δ^{min} et δ^{max}. Les résultats obtenus sont reportés sur les graphiques de la figure 8.6. Sur ces graphiques, nous avons également reporté les résultats obtenus à l'aide des formules *locales*. Les bornes de la latence sont notées $\overline{WCL}(\mathcal{L}_1)$ et $\underline{BCL}(\mathcal{L}_1)$ et sont présentées dans l'annexe C. A noter qu'une variation de δ^{min} n'a pas d'impact sur la latence pire cas et qu'une variation de δ^{max} n'a pas d'impact sur la latence meilleur cas.

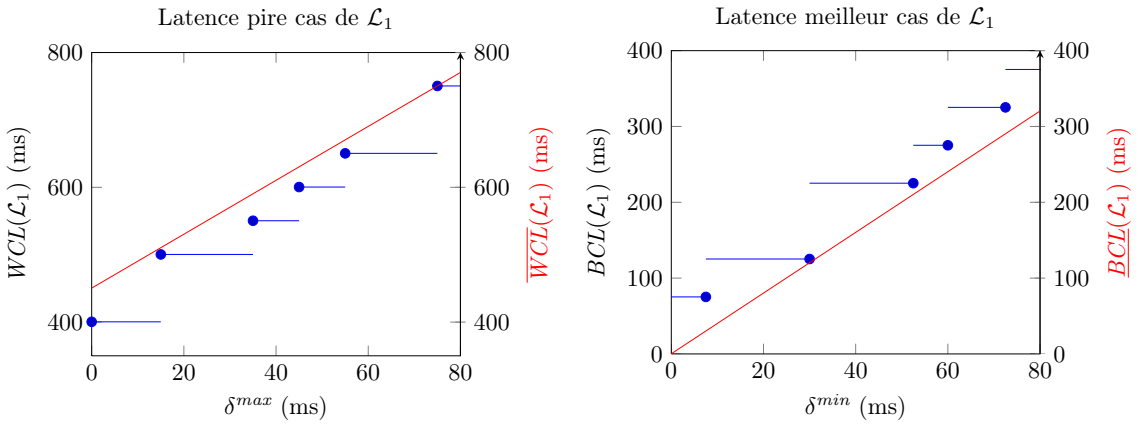


FIGURE 8.6 – Evolution des latences meilleur et pire cas en fonction des meilleurs et pires délais de communication

Sur ces graphiques, nous pouvons constater que notre méthode permet de dégager des zones d'insensibilité de la latence pire cas, par rapport aux délais de communication pire cas. Des perspectives ouvertes par ce constat seront présentées dans la section 9.2. Notre méthode permet non seulement cette observation, mais elle offre également de meilleurs résultats qu'une approche *locale*.

8.3.2 Exigence de fraîcheur

Pour l'exigence de fraîcheur $Fresness_{\leq 400}(\mathcal{F}_1)$, la fraîcheur pire cas obtenue est $WCF(\mathcal{F}_1) = 294,48$ms. Ainsi, cette exigence de fraîcheur est satisfaite. Le comportement réalisant ce pire cas est représenté sur la figure 8.7. Le pire cas de fraîcheur est mesuré entre la production de l'occurrence $pres_1[5]$ par le capteur $sensor_1$ et l'utilisation l'occurrence $disp_1[8]$ par l'actionneur $display_1$. Les occurrences des fonctions orangées correspondent aux exécutions produisant des occurrences de variable dépendant $pres_1[5]$. Les occurrences des fonctions bleutées correspondent aux premières exécutions ne produisant plus d'occurrences de variable dépendant de $pres_1[5]$. Pour qu'il n'y ait pas d'ambiguité entre les occurrences de $pres_1$ en sortie du capteur et en sortie du RDC R_1, les occurrences de $pres_1$ en sortie du RDC sont notées $pres_1_R_1[i]$. Pour ce comportement, les phases des équipements sont : $\phi_{M_1} = 0$ ms, $\phi_{M_3} = 36,54$ ms, $\phi_{M_5} = 5,956$ ms, $\phi_{R_1} = 25,82$ ms et $\phi_{sensor_1} = 5,62$ ms.

De la même manière que pour l'exigence de latence sur la chaîne $\mathcal{L}_1$, nous testons la sensibilité de l'exigence vis-à-vis des délais de communication. Pour observer cette insensibilité, nous réalisons plusieurs évaluations de $WCF(\mathcal{F}_1)$ en faisant varier δ^{max}. Les résultats obtenus sont reportés sur le graphique de la figure 8.8. A noter qu'une variation de δ^{min} n'a pas d'impact sur la fraîcheur pire cas.

Sur ce graphique, nous pouvons constater que notre méthode et la méthode *locale* offrent les mêmes résultats, les deux courbes étant confondues. La fraîcheur pire cas de la chaîne $\mathcal{F}_1$ est sensible aux variations des délais de communication pires cas. Ceci n'est pas dû à la nature de l'exigence, mais au fait que la chaîne fonctionnelle $\mathcal{F}_1$ ne passe pas plusieurs fois par un même module.

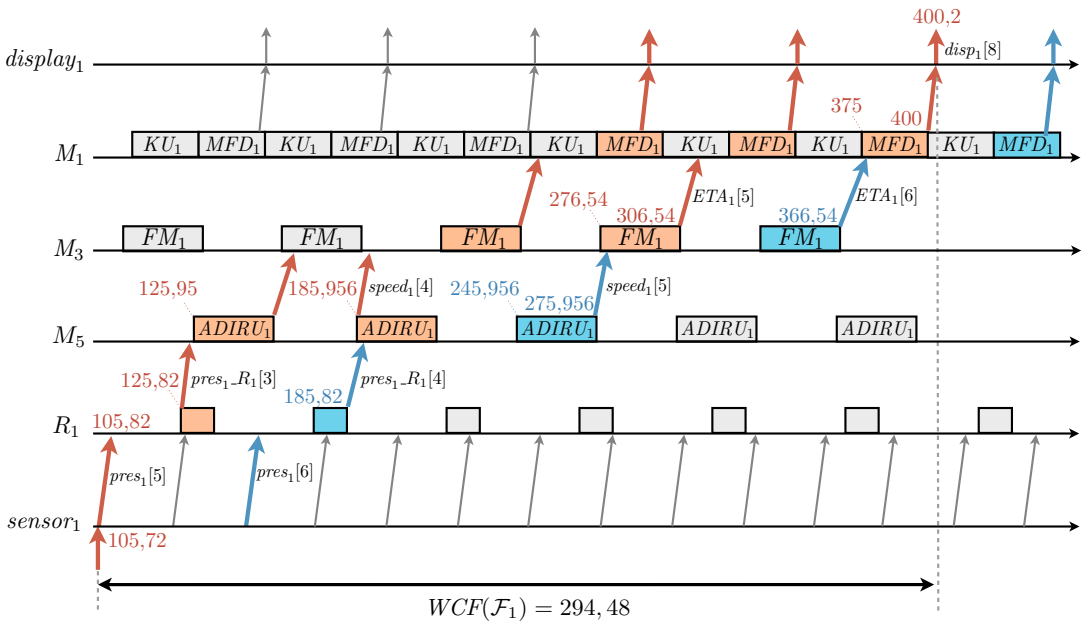


FIGURE 8.7 – Exemple d'un comportement donnant la fraîcheur pire cas de $\mathcal{F}_1$

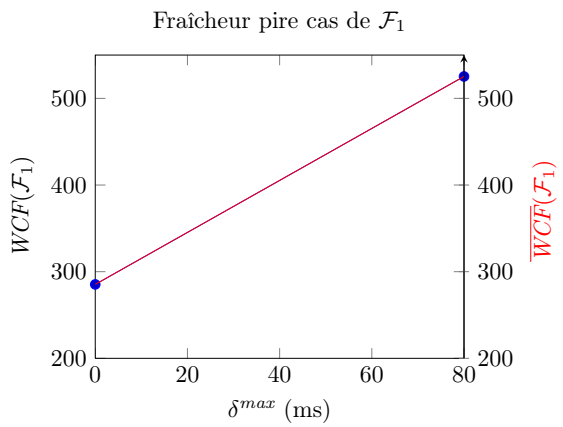


FIGURE 8.8 – Evolution de la fraîcheur pire cas en fonction du pire délai de communication

8.3.3 Exigence de cohérence entre chaînes fonctionnelles divergentes

Pour l'exigence de cohérence entre chaînes fonctionnelles divergentes $ConsistD_{\leq 500}(\mathcal{L}_1, \mathcal{L}_2)$, nous obtenons une cohérence pire cas $WCCD(\mathcal{L}_1, \mathcal{L}_2) = 353,426$ms. Ainsi, cette exigence de cohérence est satisfaite.

De la même manière que pour les exigences précédentes, nous testons la sensibilité de l'exigence de cohérence vis-à-vis des délais de communication. Pour observer cette insensibilité, nous réalisons plusieurs évaluations de $WCCD(\mathcal{L}_1, \mathcal{L}_2)$ en faisant varier δ^{max} pour un δ^{min} fixe.

Sur ces graphiques, nous pouvons constater que notre méthode offre de meilleurs résultats que la méthode *locale*. Deux facteurs expliquent cette amélioration : (1) comme vue lors de l'évaluation de la latence, notre méthode est plus précise lorsqu'une chaîne fonctionnelle traverse plusieurs fois un même module et (2) notre méthode permet de prendre en compte l'interdépendance des chaînes $\mathcal{L}_1$ et $\mathcal{L}_2$. En effet, ces chaînes traversent toutes les deux la fonction *NDB*. Les événements le long d'une des chaînes influencent donc les événements de l'autre chaîne. Prendre en compte ces dépendances permet d'affiner les résultats.

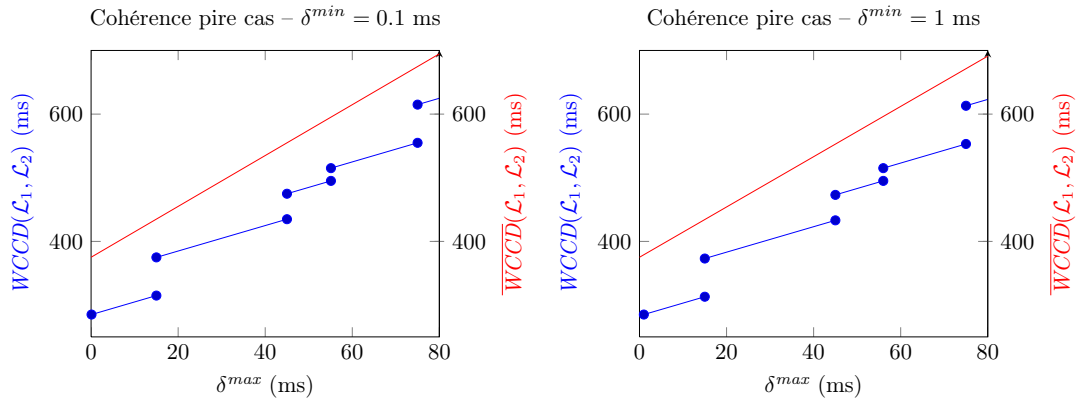


FIGURE 8.9 – Evolution de la cohérence pire cas entre $\mathcal{L}_1$ et $\mathcal{L}_2$ en fonction du pire délai de communication

8.3.4 Exigence de cohérence entre chaînes fonctionnelles convergentes

Pour l'exigence de cohérence entre chaînes fonctionnelles convergentes $ConsistC_{\leq 300}(\mathcal{F}_1, \mathcal{F}_2)$, nous obtenons une cohérence pire cas $WCCC(\mathcal{F}_1, \mathcal{F}_2) = 170,238$ms. Ainsi, cette exigence de cohérence est satisfaite. Notre méthode fournit les mêmes résultats que la méthode *locale*, dans la mesure où les deux chaînes $\mathcal{F}_1$ et $\mathcal{F}_2$ ne traversent pas plusieurs fois un même module et qu'elles sont indépendantes, c'est-à-dire qu'elles ne se croisent pas avant de converger. Nous ne représentons pas l'évolution de $WCCC(\mathcal{F}_1, \mathcal{F}_2)$ en fonction de δ^{max} et δ^{min}. La pire cohérence s'accroît linéairement en fonction de la distance entre δ^{max} et δ^{min}.

8.3.5 Conclusion

Le tableau 8.1 résume les résultats obtenus sur l'étude de cas avec la méthode fondée sur la résolution de MILP et la méthode *locale*.

Exigence	Résultats méthode *globale*		Résultats méthode *locale*	
	Pire cas	Meilleur cas	Pire cas	Meilleur cas
$Latency_{<700}(\mathcal{L}_1)$	450,4	75,2	524,132	1,358
$Fresness_{\leq 400}(\mathcal{F}_1)$	294,48	1,052	294,48	1,052
$ConsistD_{\leq 500}(\mathcal{L}_1, \mathcal{L}_2)$	353,426	0	449,032	0
$ConsistC_{\leq 300}(\mathcal{F}_1, \mathcal{F}_2)$	170,238	0	170,238	0

TABLE 8.1 – Comparatif des résultats des méthodes *globale* et *locale*

Au travers de ces expérimentations, nous avons constaté que notre méthode apporte un gain, en terme de précision des évaluations des exigences, pour certains motifs de chaînes fonctionnelles, notamment pour les chaînes traversant plusieurs fois un même module. Un deuxième avantage est de fournir un exemple de comportement menant au meilleur ou au pire cas. Cette capacité d'observation ouvre des possibilités d'amélioration pour les formules analytiques. Nous discutons ce point dans la suite.

Pour chaque expérience réalisée dans cette sous-section, la résolution de chaque MILP requiert un temps de calcul inférieur à 5 ms, à l'exception de l'exigence de cohérence entre chaînes divergentes qui requiert 85 ms. Dans la suite, nous effectuons d'autres expériences pour tester le passage à l'échelle de notre méthode.

Chapitre 9

Discussions

Dans ce chapitre, nous discutons deux points qui nous ont semblé particulièrement intéressant. Premièrement, la complexité de notre méthode est par nature exponentielle, car elle repose sur la recherche d'une solution optimale d'un MILP. Il est donc nécessaire d'évaluer expérimentalement la robustesse de la méthode vis-à-vis du passage à l'échelle. Ensuite, nous nous concentrons sur un phénomène observé sur l'étude de cas : l'évolution par palier des pires ou meilleurs cas de certaines exigences, en fonction du pire temps de communication. Nous expliquons, sur un exemple simple, l'origine de ce phénomène. Sa mise en équation ouvre des perspectives d'améliorations pour les formules *locales* que nous avons présentées précédemment. Nous appliquons alors cette idée pour une exigence de latence et proposons une formule améliorée, valable pour des chaînes fonctionnelles ayant un motif particulier.

9.1 Passage à l'échelle

La méthode que nous proposons permet d'obtenir des résultats plus précis que la méthode *locale*, mais au prix d'une complexité supérieure. Nous réalisons alors une série d'expériences afin d'évaluer si ce coût est acceptable sur des systèmes de taille industrielle. Nous discutons également le choix des options de résolution, utilisables par le solveur.

9.1.1 Expérimentations sur une étude de cas paramétrée

Nous définissons une version paramétrée de l'étude de cas, représentée sur la figure 9.1. Deux paramètres sont utilisés : N et P. Augmenter P revient à augmenter la taille de l'étude de cas en profondeur, c'est-à-dire que les chaînes fonctionnelles sont plus longues. Augmenter N revient à augmenter la taille de l'étude de cas en largeur, c'est-à-dire qu'il y aura un nombre plus important de chaînes fonctionnelles à considérer.

Les chaînes fonctionnelles considérées, pour la latence et la cohérence entre chaînes fonctionnelles divergentes, sont alors $i = 1..N$:

$$\begin{aligned}\mathcal{L}_i = {}^{req_1}\, KU_1\, {}^{wpId_1}\, FM_1\, {}^{query_{i,1}} \\ NDB_1\, {}^{query_{i,2}} \ldots {}^{query_{i,P-1}}\, NDB_{P-1}\, {}^{query_{i,P}}\, NDB_P\, {}^{answer_{i,P}} \\ NDB_{P-1}\, {}^{answer_{i,P-1}} \ldots {}^{answer_{i,2}}\, NDB_1\, {}^{answer_{i,1}} \\ FM_i\, {}^{wpInfo_i}\, MFD_i\, {}^{disp_i}\end{aligned} \tag{9.1}$$

Pour une étude de cas, avec les paramètres N et P, nous définissons ainsi N chaînes divergentes. Toutes ont la même origine : req_1. Nous évaluons alors la latence pire cas de la chaîne $\mathcal{L}_1$, c'est-à-dire $WCL(\mathcal{L}_1)$ ainsi que la cohérence pire cas entre ces chaînes divergentes, c'est-à-dire $WCCD(\mathcal{L}_1, \ldots, \mathcal{L}_N)$. Chaque chaîne $\mathcal{L}_i$ traverse $2 \cdot P + 3$ fonctions.

Les chaînes fonctionnelles considérées, pour la fraîcheur et la cohérence entre chaînes fonctionnelles convergentes, sont alors $i = 1..N$:

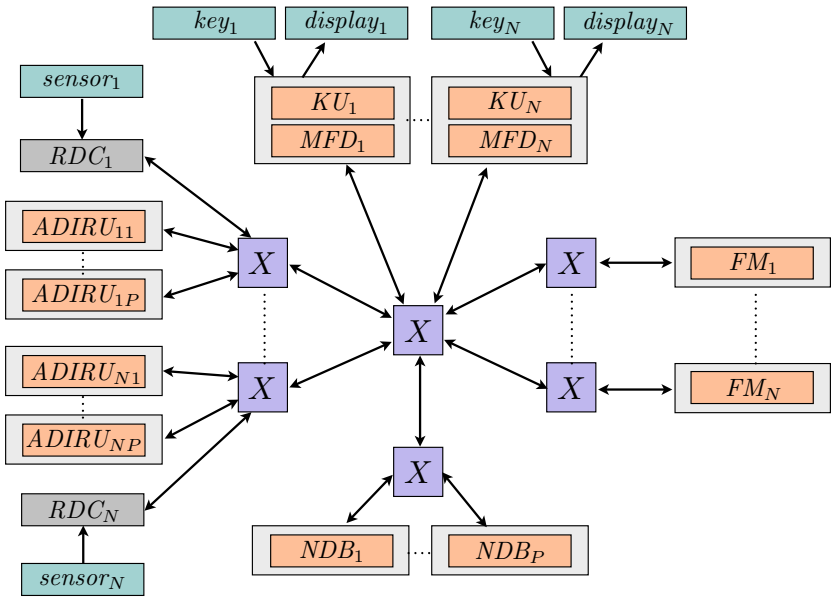


FIGURE 9.1 – Définition d'une étude de cas paramétrée

$$\mathcal{F}_i = {}^{pres_{i,1}}ADIRU_{i,1}\ {}^{pres_{i,2}}ADIRU_{i,2}\ {}^{pres_{i,3}}\ldots\ {}^{pres_{i,P}}ADIRU_{i,P}\ {}^{speed_i}FM_1\ {}^{ETA_1}MFD_1\ {}^{disp_1} \tag{9.2}$$

Pour une étude de cas, avec les paramètres N et P, nous définissons ainsi N chaînes divergentes. Toutes ont la même fin : $disp_1$. Nous évaluons la fraîcheur pire cas de la chaîne $\mathcal{F}_1$, c'est-à-dire $WCF(\mathcal{F}_1)$ ainsi que la cohérence pire cas entre ces chaînes convergentes, c'est-à-dire $WCCC(\mathcal{F}_1, \ldots, \mathcal{F}_N)$. Chaque chaîne $\mathcal{F}_i$ traverse $P + 2$ fonctions.

Sur la figure 9.2 nous représentons l'évolution du temps requis pour résoudre les MILP associés aux exigences des chaînes fonctionnelles. Pour la latence et la fraîcheur pire cas, nous faisons varier le paramètre P, ce qui a pour effet d'allonger les chaînes fonctionnelles. Pour les cohérences pire cas, nous faisons varier les paramètres N et P. Faire varier N augmente le nombre de chaînes fonctionnelles considérées dans le problème, et augmenter P allonge ces chaînes. Toutes les résolutions ont été effectuées avec le solveur `lp_solve` en version `5.5.0.13`, sur un processeur `Intel Core 2 Duo` à 2,52 GHz, sous `Mac OS X 10.6.8`.

9.1.2 Options du solveur

N'étant pas spécialistes des techniques de résolution des programmes linéaires mixtes, nous avons réalisé toutes ces expérimentations en utilisant les options par défaut du solveur. Ces options permettent de forcer l'utilisation de certains algorithmes lors de la résolution. Le nombre de combinaisons d'algorithmes utilisables est considérable, nous n'en proposons pas une étude exhaustive. Cependant, par curiosité, nous avons testé quelques variations autour des options par défaut. Deux algorithmes sont fondamentaux dans la résolution d'un MILP : l'algorithme du simplexe introduit par Dantzig dans [41] et l'algorithme de séparation et évaluation (*branch and bound*)[66]. Ces deux algorithmes décrivent des méthodes de résolution générales et peuvent être implémentés de différentes façons. Pour le simplexe, nous testons les options suivantes proposées par `lp_solve` :

- `piv0` : utilise la règle de pivotage choisissant le premier sommet adjacent,
- `piv1` : utilise la règle de pivotage définie par Dantzig,
- `piv2` : (option par défaut) utilise la règle de pivotage du *Devex pricing* définie par Paula M.J. Harris,
- `piv3` : utilise la règle de pivotage choisissant le sommet en fonction de la pente la plus forte.

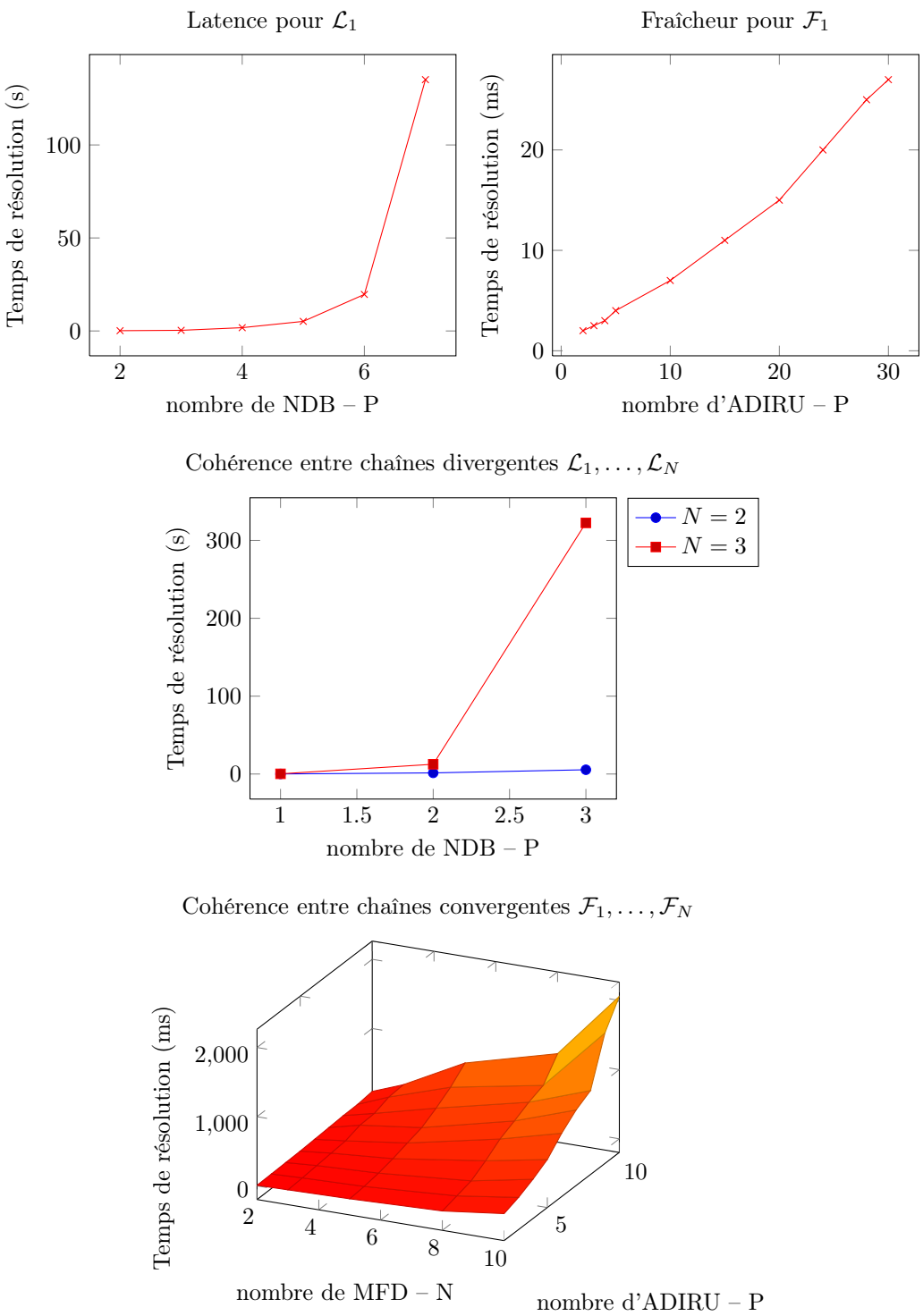


FIGURE 9.2 – Temps de résolution des programmes linéaires en nombres entiers pour la latence, la fraîcheur et les cohérences entre chaînes fonctionnelles divergentes et convergentes

Nous utilisons ces différentes options pour la résolution d'un problème de latence. Les temps de résolution sont reportés dans le tableau 9.1.

De la même manière, nous testons les options suivantes proposées par `lp_solve` pour le *branch and bound* :

– `B0` : (option par défaut) choix de la colonne de plus petit indice,

	piv0	piv1	piv2 (défaut)	piv3
Temps de résolution (s)	2,7	5	5,13	5,53

TABLE 9.1 – Variation du temps de résolution en fonction de la règle de pivotage utilisée

- B1 : choix en fonction de la distance avec la borne courante,
- B2 : choix en fonction de la plus grande borne courante,
- B3 : choix en fonction de la plus grande valeur fractionnaire,
- B4 : choix en fonction d'une fonction de coût.

Nous utilisons ces différentes options pour la résolution du même problème de latence. Toutes les autres options restent à leur valeur par défaut. Les temps de résolution sont reportés dans le tableau 9.2.

	B0 (défaut)	B1	B2	B3	B4
Temps de résolution (s)	5,13	5,41	6,1	14,6	145,6

TABLE 9.2 – Variation du temps de résolution pour différents algorithmes de *branch and bound*

Comme nous pouvons le constater, il existe des différences entre les temps de résolution en fonction des algorithmes utilisés. Il serait intéressant de déterminer l'algorithme, ou plutôt la combinaison d'algorithmes, optimale vis-à-vis de notre problème. Pour obtenir de meilleures performances, une autre perspective serait d'optimiser la formalisation du problème en fonction d'un algorithme particulier.

9.1.3 Conclusion

Sur les résultats obtenus nous pouvons observer que, pour un même nombre de fonctions traversées, une exigence peut être évaluée plus ou moins rapidement. Par exemple, l'évaluation de la fraîcheur d'une chaîne fonctionnelle composée de 14 fonctions, ce qui correspond au paramètre $P = 12$, requiert 12 ms, alors que pour la latence d'une chaîne de 13 fonctions, ce qui correspond à $P = 5$, il faut 5,13 s. Nous en déduisons que, pour le solveur, il existe des problèmes « faciles » et d'autres « difficiles ». La difficulté d'un problème ne vient pas de la nature de son exigence (latence, fraîcheur ou cohérence). Nous identifions les problèmes difficiles comme ceux dont les chaînes fonctionnelles passent plusieurs fois par un même module. Cependant, les résultats obtenus sur l'étude de cas paramétrée tendent à montrer que notre méthode est *su samment* robuste vis-à-vis du passage à l'échelle, même pour ces problèmes. Le temps requis pour la résolution « explose » en fonction de la longueur de la chaîne, mais reste raisonnable vis-à-vis du besoin industriel. Par exemple, évaluer la latence sur l'étude de cas où $P = 7$, ce qui correspond à 15 fonctions traversées, prend 120 secondes, mais il est entendu que les chaînes fonctionnelles réelles dépassent rarement une dizaine de fonctions.

Une difficulté supplémentaire apparaît sur l'exigence de cohérence entre les chaînes divergentes. Dans cet exemple, non seulement les chaînes fonctionnelles traversent plusieurs fois un même équipement, mais en plus il existe des interactions entre ces chaînes, ce qui complexifie la résolution. En outre, nous avons constaté précédemment (c.f. 8.2.3 page 127) que la traduction d'une exigence de cohérence en une fonction objectif nécessite l'ajout de variables booléennes dans le programme linéaire, ce qui en complexifie la résolution. Des idées pour améliorer les temps de résolution dans ce contexte, sont proposées dans les perspectives du manuscrit.

Un des avantages de l'utilisation d'un solveur est donc qu'il est capable de résoudre rapidement les problèmes simples. En effet, comme évoqué dans l'état de l'art (c.f. sous-section 2.2.1 page 25), nous avons testé, en première approche, une modélisation du problème avec des automates temporisés, ce qui permet l'utilisation d'un *model checker* pour la vérification. Nous n'avons pas noté de différence sensible dans le temps de résolution d'un problème facile ou d'un problème difficile. Nous avons également déroulé des expérimentations pour tester la sensibilité du temps de résolution à la différence d'échelle de temps entre les composants. Nous avons constaté que notre méthode est insensible à ces différences d'échelles, ce qui constitue un autre avantage.

9.2 Ouverture vers des formules analytiques plus précises : application à la latence

Les observations réalisées en testant notre méthode sur l'étude de cas tendent à montrer que les formules analytiques *locales* sont sources de sur-approximation pour les chaînes fonctionnelles traversant plusieurs fois un même module. Nous présentons un exemple simple illustrant ce phénomène.

9.2.1 Un exemple simple

Prenons l'exemple d'une chaîne fonctionnelle $\mathcal{F} = {}^{v_0}\ F_1\ {}^{v_1}\ F_2\ {}^{v_2}\ F_1\ {}^{v_3}$ telle que représentée sur la figure 9.3.

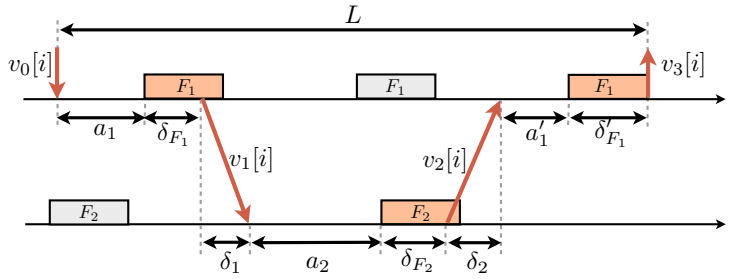


FIGURE 9.3 – Illustration du gain de la méthode

Sur cette figure, a_1 est le temps écoulé entre l'arrivée de l'occurrence $v_1[i]$ et l'activation de l'occurrence de F_1 pendant laquelle elle est traitée. En utilisant le formalisme du *tagged signal model*, nous avons : $a_1 = t_{i_F}^{F_1} - t_{i_1}^{v_1}$. a_2 et a_1 représentent les mêmes grandeurs pour la fonction F_2 et la deuxième occurrence considérée pour F_1. Les temps de traitement de ces occurrences de fonction sont représentés par δ_{F_1}, δ_{F_2} et δ_{F_1}. Les délais de communication de F_1 à F_2 et de F_2 à F_1 sont respectivement δ_1 et δ_2. La latence entre l'arrivée de $v_1[i]$ et la production de $v_3[i]$ est alors :

$$L = a_1 + \delta_{F_1} + \delta_1 + a_2 + \delta_{F_2} + \delta_2 + a_1 + \delta_{F_1} \tag{9.3}$$

La méthode *locale*, repose sur $a_1 \leq T_{F_1}$, mais cet encadrement est potentiellement une sur-approximation : du fait de la périodicité de F_1, a_1 est également contraint par l'instant d'activation de l'occurrence de F_1, ayant traité $v_1[i]$. Plus précisément, il existe un entier $k \quad \mathbb{N}^*$ tel que :

$$\delta_{F_1} + \delta_1 + a_2 + \delta_{F_2} + \delta_2 = k \cdot T_{F_1} \tag{9.4}$$

où k est le plus petit nombre de périodes entières, séparant l'occurrence de F_1, ayant traité $v_1[i]$, et l'arrivée de $v_2[i]$ dans le module de F_1. La latence s'exprime alors :

$$\begin{aligned} L &= a_1 + k \cdot T_{F_1} + \delta_1 \\ \text{avec : } k &= \left\lceil \frac{\delta_{F_1} + \delta_1 + a_2 + \delta_{F_2} + \delta_2}{T_{F_1}} \right\rceil \end{aligned} \tag{9.5}$$

Avant d'être traitée par F_1, $v_0[i]$ attend au maximum une période de F_1, ainsi : $a_1 \leq T_{F_1}$. De même, $v_1[i]$ attend au maximum une période de F_2 avant son traitement, ainsi : $a_2 \leq T_{F_2}$. Les temps de traitement de F_1 et F_2 sont bornés par : $\delta_{F_2} \leq C_{F_2}$ et $\delta_{F_2} \leq C_{F_2}$. Nous supposons également que les délais de communication maximaux entre F_1 et F_2 et entre F_2 et F_1 sont respectivement δ_1^{max} et δ_2^{max}. Nous disposons ainsi d'un encadrement de chacun des termes de la latence. Comme la fonction de la partie entière supérieure $x \quad x$ est croissante, nous pouvons déduire un encadrement supérieur de la latence, noté L_{max} :

$$\begin{aligned} L &\leq L_{max} \\ L_{max} &= T_{F_1} + \left\lceil \frac{C_{F_1} + \delta_1^{max} + T_{F_2} + C_{F_2} + \delta_2^{max}}{T_{F_1}} \right\rceil \cdot T_{F_1} + C_{F_1} \end{aligned} \tag{9.6}$$

Du fait de la présence de la partie entière dans la définition de L_{max}, cette borne évolue par palier en fonction des pires temps de communication δ_1^{max} et δ_2^{max}. Sur la figure 9.4 nous représentons la valeur de L_{max} obtenue en faisant varier ces délais. Les valeurs numériques utilisées pour les fonctions F_1 et F_2 correspondent aux paramètres des fonctions KU_1 et FM_1 de l'étude de cas. On retrouve alors dans l'évolution de L_{max} certaines observations faites sur l'évolution de la latence pire cas de la chaîne $\mathcal{L}_1$ de l'étude de cas (c.f. figure 8.9 page 135), c'est-à-dire une évolution par palier.

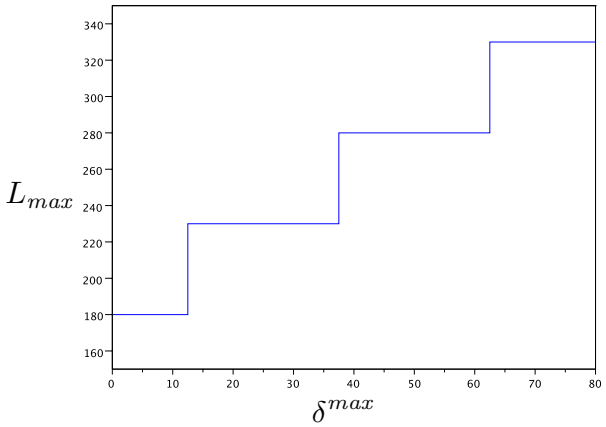


FIGURE 9.4 – Evolution de L_{max} en fonction de δ^{max}

Cependant, l'évolution de $WCL(\mathcal{L}_1)$ est plus irrégulière. En effet, $\mathcal{L}_1$ traverse deux fois le module M_1, par les fonctions KU_1 et MFD_1, et deux fois le module M_3 par la fonction FM_1. Le phénomène observé sur cet exemple simple apparaît donc deux fois le long de $\mathcal{L}_1$: une première fois sur la sous-chaîne ${}^{wpId_1}\ FM_1\ {}^{query_1}\ NDB\ {}^{answer_1}\ FM_1\ {}^{wpInfo_1}$, et une deuxième fois entre KU_1, cette sous-chaîne et MFD_1. $\mathcal{L}_1$ a une structure particulière que l'on appelle *emboîtée*. Dans la suite, nous généralisons la formule de L_{max} construite sur cet exemple simple, aux chaînes fonctionnelles emboîtées.

9.2.2 Extension aux chaînes fonctionnelles emboîtées

Dans un premier temps, nous définissons la notion de chaînes fonctionnelles emboîtées, pour ensuite définir une borne améliorée de la latence sur une chaîne de ce type. Finalement, nous appliquerons ce résultat sur la chaîne fonctionnelle emboîtée de l'étude de cas : $\mathcal{L}_1$.

Chaîne fonctionnelle emboîtée. Une chaîne fonctionnelle $\mathcal{C}$ est dite *emboîtée* si elle est de la forme :

$$\mathcal{C} = {}^{v_0}\ F_1\ {}^{v_1}\ \chi\ {}^{v_2}\ F_2\ {}^{v_3} \tag{9.7}$$

avec F_1 et F_2 deux fonctions s'exécutant sur un même module M, et χ pouvant être une chaîne fonctionnelle composé d'une unique fonction ou une chaîne fonctionnelle emboîtée.

Exemple 14 (Chaîne fonctionnelle $\mathcal{L}_1$ emboitée)**.** *Le chaîne fonctionnelle $\mathcal{L}_1$ est telle que :*

$$\begin{aligned} \mathcal{L}_1 &= {}^{req_1}\ KU_1\ {}^{wpId_1}\ \chi_1\ {}^{wpInfo_1}\ MFD_1\ {}^{disp_1} \\ \textit{avec : } \chi_1 &= {}^{wpId_1}\ FM_1\ {}^{query_1}\ \chi_2\ {}^{answer_1}\ FM_1\ {}^{wpInfo_1} \\ \textit{et : } \chi_2 &= {}^{query_1}\ NDB\ {}^{answer_1} \end{aligned} \tag{9.8}$$

Les fonctions KU_1 et MFD_1 s'exécutent toutes les deux sur le module M_1, FM_1 s'exécute sur M_3 et NDB sur M_7. La chaîne $\mathcal{L}_1$ est donc une chaîne fonctionnelle emboîtée.

Principe. Connaissant une borne de la latence pire cas de χ, $\overline{WCL}(\chi)$, il est possible d'utiliser le même raisonnement que celui de l'exemple simple, pour déterminer un encadrement de la latence de $\mathcal{C}$, $\overline{WCL}(\mathcal{C})$. La difficulté est alors de déterminer quelle est l'occurrence de F_1 amenant au pire cas. Cette idée est présentée sur la figure 9.5. L_1 et L_2 sont deux latences mesurées en supposant que le pire cas est réalisé sur la sous-chaîne χ, qui n'est pas représentée. Nous pouvons remarquer que suivant l'occurrence de F_1 utilisée, ces latences varient. Dans la suite, nous formalisons la recherche de l'occurrence de F_1 donnant la latence pire cas.

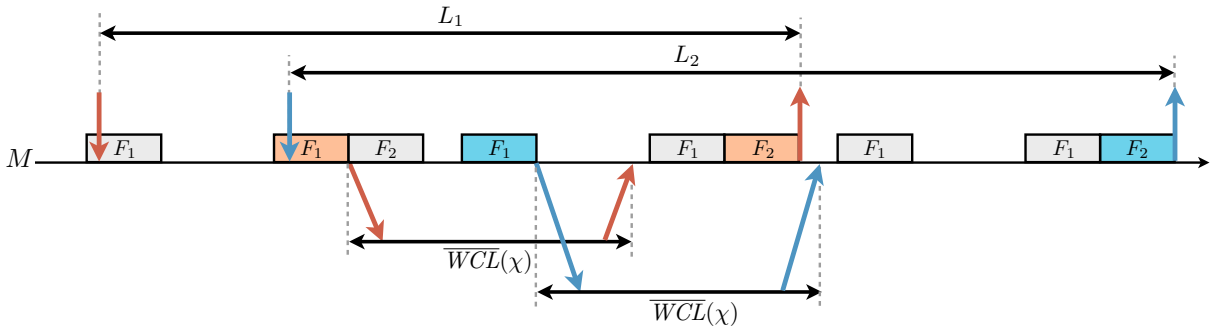


FIGURE 9.5 – Extension aux chaînes emboîtées

Recherche d une borne améliorée pour une chaîne emboîtée. Si F_1 et F_2 sont deux fonctions différentes, la formule de l'exemple simple ne peut s'appliquer. Nous notons δ_χ le temps écoulé entre la production de l'occurrence de v_1 en sortie de F_1 et la disponibilité de v_2 pour F_2. De la même manière qu'avec l'exemple simple, nous cherchons à maximiser le délai séparant les deux activations des fonctions. Avec le formalisme du *tagged signal model*, ce délai s'exprime :

$$\begin{aligned} t_j^{F_2} - t_i^{F_1} &= \phi_M + O_2 + j \cdot T_2 - (\phi_M + O_1 + i \cdot T_1) \\ &= O_2 - O_1 + j \cdot T_2 - i \cdot T_1 \end{aligned} \tag{9.9}$$

Le temps d'attente de l'occurrence de v_2 avant son traitement dans F_2 s'exprime :

$$a_2 = t_j^{F_2} - t_i^{F_1} - \delta_1 - \delta_\chi \tag{9.10}$$

L'occurrence de v_2 ne peut attendre plus de T_2 avant d'être traitée par F_2, nous avons donc :

$$0 \leq t_j^{F_2} - t_i^{F_1} - \delta_1 - \delta_\chi < T_2 \tag{9.11}$$

Avec l'expression (9.9), nous pouvons alors déduire une expression de j en fonction de i. En effet :

$$\begin{aligned} 0 &\leq O_2 - O_1 + j \cdot T_2 - i \cdot T_1 - \delta_1 - \delta_\chi < T_2 \\ -j \cdot T_2 &\leq O_2 - O_1 - i \cdot T_1 - \delta_1 - \delta_\chi < (1-j) \cdot T_2 \\ j &\geq \frac{O_1 - O_2 + i \cdot T_1 + \delta_1 + \delta_\chi}{T_2} > (j-1) \\ \text{comme } j \quad \mathbb{N},\ j &= \left\lceil \frac{O_1 - O_2 + i \cdot T_1 + \delta_1 + \delta_\chi}{T_2} \right\rceil^+ \end{aligned} \tag{9.12}$$

avec $x \quad \mathbb{R}$, $x^+ = \max\ 0; x$. On en déduit :

$$t_j^{F_2} - t_i^{F_1} = O_2 - O_1 + \left\lceil \frac{O_1 - O_2 + i \cdot T_1 + \delta_1 + \delta_\chi}{T_2} \right\rceil^+ \cdot T_2 - i \cdot T_1 \tag{9.13}$$

Indépendamment de la valeur de i, nous pouvons borner cette expression à l'aide des valeurs maximales de δ_1 et de δ_χ, qui sont respectivement le pire temps d'exécution de F_1, c'est-à-dire C_{F_1}, et la latence pire cas de la chaîne χ, c'est-à-dire $\overline{WCL}(\chi)$. Nous avons :

$$t_j^{F_2} - t_i^{F_1} \leq O_2 - O_1 + \left\lceil \frac{O_1 - O_2 + i \cdot T_1 + C_{F_1} + \overline{WCL}(\chi)}{T_2} \right\rceil^+ \cdot T_2 - i \cdot T_1 \tag{9.14}$$

Il s'agit alors de déterminer une valeur de i maximisant cette expression. Ce qui revient à déterminer une occurrence de F_1 réalisant le pire cas. Considérons la fonction f telle que :

$$\begin{array}{llll} f: & \mathbb{N} & \mathbb{R} \\ & k & a + \left\lceil \frac{b+k\cdot T_1}{T_2} \right\rceil^+ \cdot T_2 - k \cdot T_1 \end{array} \tag{9.15}$$

avec $a = O_2 - O_1$ et $b = O_1 - O_2 + C_{F_1} + \overline{WCL}(\chi)$. Nous cherchons à maximiser f. Pour limiter le domaine de valeurs de k à tester, nous montrons que f est périodique de période $ppcm(T_1, T_2)/T_1$, à partir de $-b/T_1$ $^+$. Soit k $\mathbb{N}$ tel que $k \geq -b/T_1$, nous avons :

$$\begin{aligned} f\left(k + \frac{ppcm(T_1,T_2)}{T_1}\right) &= a + \left\lceil \frac{b + \left(k + \frac{ppcm(T_1,T_2)}{T_1}\right) \cdot T_1}{T_2} \right\rceil^+ \cdot T_2 - \left(k + \frac{ppcm(T_1,T_2)}{T_1}\right) \cdot T_1 \\ &= a + \left\lceil \frac{b + k \cdot T_1}{T_2} + \frac{ppcm(T_1,T_2)}{T_2} \right\rceil^+ \cdot T_2 - k \cdot T_1 - ppcm(T_1,T_2) \end{aligned}$$

Comme $k \geq -b/T_1$, nous avons $(b + k \cdot T_1)/T_2 \geq 0$ et par définition, $ppcm(T_1, T_2)/T_2$ est un entier positif, nous pouvons donc le sortir de la partie entière :

$$\begin{aligned} f\left(k + \frac{ppcm(T_1,T_2)}{T_1}\right) &= a + \left\lceil \frac{b + k \cdot T_1}{T_2} \right\rceil^+ \cdot T_2 + \frac{ppcm(T_1,T_2)}{T_2} \cdot T_2 - k \cdot T_1 - ppcm(T_1,T_2) \\ &= a + \left\lceil \frac{b + k \cdot T_1}{T_2} \right\rceil^+ \cdot T_2 - k \cdot T_1 \\ &= f(k) \end{aligned}$$

Ainsi, pour déterminer le maximum de f nous pouvons nous limiter à :

$$k \quad \left[\left\lceil \frac{-b}{T_1} \right\rceil^+, \left(\frac{ppcm(T_1,T_2)}{T_1} - 1 \right) \right] \tag{9.16}$$

Dans la suite, nous notons $\mathcal{K}$ cet intervalle. Nous avons alors une borne sur le terme $t_j^{F_2} - t_i^{F_1}$:

$$t_j^{F_2} - t_i^{F_1} \leq \max_{k \ \mathcal{K}} \left\{ O_2 - O_1 + \left\lceil \frac{O_1 - O_2 + k \cdot T_1 + C_{F_1} + \overline{WCL}(\chi)}{T_2} \right\rceil^+ \cdot T_2 - k \cdot T_1 \right\} \tag{9.17}$$

Cette borne nous permet finalement de majorer la latence de la chaîne $\mathcal{C}$.

Borne supérieure améliorée de la latence. Pour une chaîne fonctionnelle emboîtée $\mathcal{C}$, de sous-chaîne χ telle que $\mathcal{C} =^{v_0} F_1 \ ^{v_1} \chi \ ^{v_2} F_2 \ ^{v_3}$, une borne supérieure de la latence est donnée par :

$$\begin{aligned} &\overline{WCL}(\mathcal{C}) = T_1 + O_2 - O_1 + C_2 + \max_{k \ \mathcal{K}} \left\{ \left\lceil \frac{O_1 - O_2 + k \cdot T_1 + C_{F_1} + \overline{WCL}(\chi)}{T_2} \right\rceil^+ \cdot T_2 - k \cdot T_1 \right\} \\ &\text{avec : } \mathcal{K} = \left[\left\lceil \frac{-b}{T_1} \right\rceil^+, \left(\frac{ppcm(T_1,T_2)}{T_1} - 1 \right) \right] \end{aligned} \tag{9.18}$$

Le calcul de cette borne peut être simplifié, dans le cas où la période T_2 est un multiple de T_1, c'est-à-dire h $\mathbb{N}$ tel que $T_1 = h \cdot T_2$, et si le terme $b = O_1 - O_2 + C_{F_1} + \overline{WCL}(\chi)$ introduit dans la fonction f précédemment est positif. En effet, si ces hypothèses sont vérifiées, nous avons k $\mathbb{N}$:

$$\begin{aligned} f(k) &= a + \left\lceil \frac{b + k \cdot T_1}{T_2} \right\rceil^+ \cdot T_2 - k \cdot T_1 \\ &= a + \left\lceil \frac{b + k \cdot h \cdot T_2}{T_2} \right\rceil^+ \cdot T_2 - k \cdot h \cdot T_2 \\ &= a + \left\lceil \frac{b}{T_2} \right\rceil^+ \cdot T_2 + k \cdot h \cdot T_2 - k \cdot h \cdot T_2 \\ &= a + \left\lceil \frac{b}{T_2} \right\rceil^+ \cdot T_2 \end{aligned} \tag{9.19}$$

Ainsi, la borne de la latence ne dépend plus de l'occurrence réalisant la pire cas, et nous avons :

$$\overline{WCL}(\mathcal{C}) = T_1 + C_2 + O_2 - O_1 + \left\lceil \frac{O_1 - O_2 + C_{F_1} + \overline{WCL}(\chi)}{T_2} \right\rceil^+ \cdot T_2 \tag{9.20}$$

A présent, nous appliquons ce résultat à la chaîne fonctionnelle $\mathcal{L}_1$ de l'étude de cas.

Application à la chaîne fonctionnelle $\mathcal{L}_1$. Comme introduit dans l'exemple 14, $\mathcal{L}_1$ est une chaîne fonctionnelle emboîtée. Nous exprimons donc les bornes de la latence de ses sous-chaînes, en commençant par celle de plus-bas niveau, χ_2 et en remontant vers la chaîne complète.

La sous-chaîne χ_2 n'étant composée que d'une unique fonction, nous calculons $\overline{WCL}(\chi_2)$ à l'aide de la méthode *locale* présentée à la sous-section C page 161.

$$\overline{WCL}(\chi_2) = (N_{VL_4} - 1) \cdot bag_{VL_4} + b_{C_4} + T_{NDB} + C_{NDB} + (N_{VL_7} - 1) \cdot bag_{VL_7} + b_{C_7} \tag{9.21}$$

La sous-chaîne χ_1 part de FM_1, puis passe par χ_1 pour revenir à FM_1. En outre, comme $O_{FM_1} - O_{FM_1} + C_{FM_1} + \overline{WCL}(\chi_1) \geq 0$, nous pouvons utiliser la formule (9.20). A cette formule, nous ajoutons le temps de traversée pire cas des variables liant χ_1 à $\mathcal{L}_1$, ainsi :

$$\begin{aligned} \overline{WCL}(\chi_1) = & T_{FM_1} + C_{FM_1} + \left\lceil \frac{C_{FM_1} + \overline{WCL}(\chi_2)}{T_2} \right\rceil^+ \cdot T_2 \\ & + (N_{VL_1} - 1) \cdot bag_{VL_1} + b_{C_1} \\ & + (N_{VL_3} - 1) \cdot bag_{VL_3} + b_{C_3} \end{aligned} \tag{9.22}$$

De la même manière nous pouvons utiliser la formule (9.20), en ajoutant les latences dues au capteur key_1 et à l'actionneur $disp_1$, pour exprimer une borne supérieure de la latence de $\mathcal{L}_1$:

$$\begin{aligned} \overline{WCL}(\mathcal{L}_1) = & T_{KU_1} + C_{MFD_1} + O_{MFD_1} - O_{KU_1} + \left\lceil \frac{C_{FM_1} + \overline{WCL}(\chi_1)}{T_2} \right\rceil^+ \cdot T_2 \\ & + b_{key_1} + b_{disp_1} \end{aligned} \tag{9.23}$$

Cette formule donne les mêmes résultats que la méthode par résolution de programmes linéaires.

Conclusion. Les observations réalisées, grâce à la méthode par résolution de programmes linéaires, nous ont inspiré la recherche de formules analytiques plus précises que celles de la méthode *locale*. Nous avons présenté une formule donnant une borne pour la latence pour une chaîne fonctionnelle avec un motif particulier. Une question est alors ouverte sur la possibilité de définir ce genre de formules pour d'autres motifs de chaînes fonctionnelles et pour les autres types d'exigences.

Conclusion et perspectives

Conclusion

Nous avons exploré la problématique de la vérification d'exigences temps réel, spécifiquement dans le cadre des systèmes dits *Avioniques Modulaires Intégrées*. Ces systèmes sont caractérisés par leur architecture distribuée, asynchrone et dont la configuration est statique. Des fonctionnalités de plus en plus complexes sont implémentées sur ces systèmes, il est alors nécessaire de disposer de méthodes de vérification précises et efficaces. La précision des méthodes permet alors de dimensionner au mieux le système.

Une part importante de notre travail a consisté à formaliser le problème posé. Nous avons opté pour le formalisme du *tagged signal model*. Notre première contribution a été de proposer une modélisation du comportement des constituants d'un système IMA. Les comportements possibles du système global sont alors définis par composition. Ceci nous permet finalement de définir dans ce formalisme la sémantique des exigences temps réel que doit satisfaire ce modèle.

Le notion de *chaîne fonctionnelle* est centrale dans notre approche, car elle permet d'avoir une vision de *bout en bout* du système. Nous considérons que cette prise en compte de *bout en bout* est nécessaire pour obtenir des résultats précis et représentatifs du comportement réel du système. Autour de cette notion de chaîne fonctionnelle, trois catégories d'exigences nous ont paru pertinentes : l'exigence de latence, qui est la notion la plus classique, et les exigences de fraîcheur et de cohérence, qui sont plus rares dans la littérature. La satisfaction de ces exigences a été exprimée dans le formalisme utilisé pour modéliser le système. Un deuxième résultat de notre thèse est l'exploitation de ce modèle pour la vérification des exigences. En effet, le *tagged signal model* étant un formalisme très général, il n'existe pas d'outils permettant d'analyser notre modèle. Cependant, la modélisation que nous avons choisi repose sur la définition de contraintes entre les événements du système. Pour chaque exigence, nous proposons alors d'extraire du modèle les contraintes qui lui sont relatives, c'est-à-dire les contraintes liant un événement en entrée de la chaîne fonctionnelle de l'exigence jusqu'à un événement dépendant en sortie, ainsi que tous les événements dépendant de ces deux événements. Ensuite, ces contraintes sont utilisées pour définir un programme linéaire en nombres entiers, dont la solution optimale est le pire ou le meilleur cas de l'exigence. Nous pouvons alors utiliser un solveur de programmes linéaires, pour vérifier la satisfaction de l'exigence. Deux difficultés ont dû être levées :

- la complexité du modèle rend inutilisable cette approche en pratique. En effet, sur un système IMA le réseau de communication est partagé entre toutes les fonctions. Ainsi en recherchant les événements relatifs à une communication, il y a un « risque » de récupérer les événements d'une grande partie du système. Nous avons alors défini une méthode d'abstraction permettant de casser cette complexité en limitant les dépendances entre les événements. Nous mettons ainsi en œuvre notre approche sur un système abstrait.
- les expressions des exigences et de certaines contraintes ont dû être linéarisées pour permettre leur utilisation avec un solveur de programmes linéaires.

Cette approche offre une vision globale du système, et l'ensemble des comportements explorés par le solveur sont réalisables par le système abstrait. Les résultats de notre méthode ont été comparés à une approche *locale* du problème, c'est-à-dire fondée sur une analyse locale des composants du système indépendamment du contexte global. Cette analyse peut mener à considérer des scénarios impossibles, ce qui entraine des sur-approximations dans les résultats.

Nous avons mis au point un outil de calcul pour valider notre approche. En particulier, cela nous a permis de tester la robustesse de notre méthode vis-à-vis du passage à l'échelle. Bien que notre méthode de résolution soit d'une complexité exponentielle, la méthode est raisonnable sur des configurations de taille réaliste, c'est-à-dire équivalentes à celle du système de l'A380. En sortie de notre outil nous pouvons

récupérer un scénario du pire ou du meilleur cas d'une exigence. En perspective, l'analyse de ces résultats nous a inspiré des améliorations possibles de l'approche *locale*. Nous avons démontré ce point sur l'exigence de latence.

Perspectives

Généralisation et extension de l'approche

La démarche que nous avons proposé est centrée sur un modèle particulier d'application : l'avionique modulaire intégrée. Une question naturelle concerne donc les possibilités de généralisation de d'extension de l'approche. Nous donnons quelques pistes dans cette optique.

Fonction. Une première piste d'extension de l'approche serait de proposer une modélisation plus fine du comportement des fonctions. Deux axes peuvent être considérés : (1) l'enrichissement du comportement des fonctions pour, par exemple, pouvoir prendre en compte des fonctions nécessitant plusieurs périodes pour produire une sortie dépendant d'une certaine entrée, et (2) augmenter le « niveau de détail » des fonctions en prenant en compte les processus (au sens système, et non pas du *tagged signal model*) réalisant la fonction. La difficulté serait alors de prendre en compte l'ordonnancement local de ces processus dans la partition de la fonction, dans le contexte du système global. Il n'est pas garanti que la complexité d'une telle description puisse être maîtrisée.

Partitions à plusieurs tranches de temps . Dans la modélisation d'un système IMA que nous proposons, le temps alloué à une partition est défini comme un unique intervalle se répétant périodiquement. Cependant, dans le standard ARINC653, le temps alloué à une partition peut être défini comme un ensemble d'intervalle. Bien qu'en pratique, il semble que cette possibilité ne soit pas exploitée, un modèle plus réaliste devrait en proposer la modélisation.

Ressources non-IMA. Notre méthode s'applique à des fonctions allouées sur des modules IMA et qui n'interagissent qu'avec des fonctions elles-mêmes allouées sur ce genre de modules. Cependant, sur les systèmes actuels comme celui de l'A380, ces fonctions sont susceptibles d'interagir avec des calculateurs « historiques » qui ne sont pas conformes à l'ARINC653. Nous avons initié un travail pour prendre en compte des ressources de calcul dans lesquels les fonctions sont ordonnancées avec une politique de service à priorité fixe.

Ouverture vers d autres modèles Une collaboration avec les équipes du CEA-LIST, partenaire du projet SATRIMMAP, a été ouverte pour adapter notre approche à la plate-forme OASIS [42]. En particulier, l'objectif est de tirer profit des marges dégagées au niveau des exigences, par la prise en compte globale des chaînes fonctionnelles, pour distribuer au mieux les ressources de la plate-forme.

Application à l'ingénierie de systèmes

Du point de vue l'ingénierie de systèmes, notre méthode se place à la fin du cycle de conception, une fois que toutes les ressources de calcul et de communication ont été allouées. Si certaines exigences temps réel ne sont pas satisfaites, nous pouvons quantifier le dépassement de leurs bornes et fournir au concepteur des exemples d'exécution menant à ces dépassements. Ces données pourront alors lui servir de base pour modifier le système, et le rendre conforme. Dans ce contexte, il serait intéressant de réfléchir aux informations les plus pertinentes à retourner au concepteur.

Nous pouvons également envisager une autre utilisation de notre méthode. Dans le projet SATRIMMAP, l'allocation des ressources s'effectuent en deux étapes [5]. Dans un premier temps, les fonctions sont allouées sur les modules avec pour objectif d'optimiser leur utilisation, tout en respectant des contraintes de sûreté, telles que « deux fonctions redondantes doivent s'exécuter sur des modules différents et répartis de part et d'autre de l'appareil ». Une fois l'allocation des ressources de calcul terminée, les ressources de communication sont allouées de telle manière que les besoins en bande passante des fonctions soient assurés, et que le réseau ne soit pas congestionné. Un autre critère important de cette allocation est, pour chaque lien virtuel, le temps de traversée pire cas du réseau. Nous avons montré (c.f. section 8.3 page 131) que les exigences de certaines chaînes fonctionnelles sont insensibles à ce paramètre sur certaines plages

de valeurs. En d'autres termes, pour certaines chaînes fonctionnelles, il est inutile de chercher à améliorer les temps de traversée du réseau, vis-à-vis des exigences de *bout en bout*. Il serait intéressant de considérer les avantages, au niveau de la conception du système, qui pourraient découler de cette observation.

Performance de la résolution

Comme nous l'avons vu dans la discussion sur la robustesse de notre méthode vis-à-vis du passage à l'échelle (c.f. sous-section 9.1 page 137), le temps de résolution d'un MILP est raisonnable pour des chaînes fonctionnelles de taille réaliste. Cependant par nature, cette méthode a une complexité exponentielle, il est alors intéressant de considérer les options envisageables pour ménager cette complexité.

Choix des algorithmes de résolution. Ce point a déjà été abordé dans la discussion sur le passage à l'échelle. Nous avons pu constater qu'il existe des différences entre les temps de résolution, suivant les algorithmes choisis. La détermination des algorithmes les plus efficaces pour notre problème est alors une piste d'amélioration. En outre, nous avons utilisé le solveur gratuit `lp_solve`, mais d'autres solveurs commerciaux plus performant existent (`CPLEX`, `MOSEK`,...). Bien que ces optimisations ne permettront pas de casser la complexité du problème, nous pouvons espérer diviser par deux le temps de résolution.

Simpli cation du problème. Considérons l'évaluation d'une exigence de cohérence entre chaînes fonctionnelles divergentes ou cohérentes. Le solveur doit considérer chaque chaîne comme étant potentiellement celle subissant le temps de traversée le plus long, et idem pour le temps de traversée le plus court. La combinatoire du problème peut alors devenir importante, ce qui peut être rédhibitoire si les temps de traversée sont eux mêmes difficiles à obtenir. L'idée serait alors d'utiliser des propriétés structurelles du problème, comme des symétries, pour simplifier le problème, en « disqualifiant » certaines chaînes. Une chaîne disqualifiée ferait toujours partie du programme linéaire, et donc son impact sur les autres chaînes serait toujours pris en compte, mais elle ne serait plus considérée pour le meilleur ou le pire temps de traversée.

Prenons par exemple la cohérence entre les chaînes divergentes $\mathcal{L}_1, \ldots, \mathcal{L}_N$ de l'étude de cas paramétrée (c.f. sous-section 9.1 page 137). Toutes les chaînes, $\mathcal{L}_i$ pour $i = 1..N$ sont similaires : elles ont toutes pour origine la requête req_1 et finissent par leur affichage respectif $disp_i$. En outre, les éléments traversés par ces chaînes ont les mêmes caractéristiques temporelles. En revanche, la chaîne $\mathcal{L}_1$ se distingue car la première et la dernière fonction traversée s'exécutent sur un même module. Toutes les chaînes $\mathcal{L}_i$ pour $i = 2..N$ ont potentiellement le même comportement temporel. La cohérence pire cas est donc subit soit entre $\mathcal{L}_1$ et une des autres chaînes, soit entre deux des autres chaînes. Il est donc envisageable d'exprimer la cohérence seulement entre trois chaînes : $\mathcal{L}_1$ et deux des autres chaînes, ce qui réduit la combinatoire du problème.

Annexe A

Langage de description d architecture du prototype

La syntaxe détaillée du langage utilisé en entrée de notre prototype a été rapportée en annexe afin de ne pas alourdir la lecture du manuscrit. A la suite de la présentation du langage, nous présentons son utilisation à l'aide d'exemples tirés de l'étude de cas.

Syntaxe du langage

Nous donnons la syntaxe complète du langage du plus haut niveau hiérarchique, c'est-à-dire le système, vers le plus bas niveau qui est constitué des expressions élémentaires. La syntaxe est décrite à l'aide du métalangage EBNF et elle reprend la structure des modèles d'architectures décrits au chapitre 4 (page 61).

Système global

Un système global est défini comme la concaténation de la description d'une architecture fonctionnelle, d'une architecture matérielle, d'une allocation et de l'ensemble des exigences devant être respectées par le système.

```
system ::= archFunc archMat alloc requirements;
```

Architecture fonctionnelle

```
archFunc ::= (var | func | dep)+;
```

Variable

```
var ::= "var"  identifier  nature size "end" ";" ;
```

Fonction

```
func ::= "func"  identifier  periode WCET  input output "end" ";" ;
input ::= "input"  identifier  ( ","  identifier )*";";
output ::= "output"  identifier  ( ","  identifier )*";";
```

Dépendance

```
dep ::= "dep" "(" ( identifier  |  varExt ) ","
( identifier  |  varExt ) ")" "end" ";" ;
```

Divers

```
// syntaxe de la variable représentant l environnement extérieur
varExt ::= "varExt" ;

// syntaxe pour une taille de variable
size ::= "size"  nat  uniteData ";" ;

// syntaxe d une nature de variable ou de capteur (sporadique/périodique)
nature::= "nat" typeNature ";" ;

// syntaxe d une période
periode ::= "periode"  nat   uniteTps ";" ;

// syntaxe d un temps maximum d exécution
WCET ::= "WCET"  nat   uniteTps  ";" ;
```

Architecture matérielle

```
archMat ::= (module | switch | link | RDC | sensor | actuator )+ ;
```

Port

```
port ::= "port"  identifier  debit "end" ";" ;
```

Module

```
module ::= "module"  identifier  port "end" ";" ;
```

Commutateur

```
switch ::= "switch"  identifier  lag (port)+ "end" ";" ;
lag ::= "lag"  nat   uniteTps  ";" ;
```

Lien

```
link ::= "link"  identifier  "(" identifier "," identifier ")" "end" ";" ;
```

Remote Data Concentrator

```
RDC ::= "RDC"  identifier  periode WCET port "end" ";" ;
```

Capteur

```
sensor ::= "sensor"  identifier  nature periode TT  identifier  ";" "end" ";" ;
```

Actionneur

```
actuator ::= "actuator"  identifier  TT  identifier  ";" "end" ";" ;
```

Divers

```
// syntaxe d un débit
debit ::= "debit"  nat   uniteDebit  ";" ;

// syntaxe d un intervalle de temps de traversée
TT ::= "TT" intervalle ";" ;
intervalle ::= "[" nat "," nat "]"   uniteTps  ;
```

Allocation

Une allocation est constituée d'une allocation des fonctions `allocF`, d'une allocation des variables sources `AllocS`, d'une allocation des variables puits `AllocP`, d'un ensemble de 1 à n liens virtuels `(VL)+` et finalement d'une allocation des dépendances `AllocD`.

```
alloc ::= allocF allocS allocP (VL)+ allocD ;
```

Allocation des fonctions

```
allocF ::= "allocF" ( identifier  "->" partition)
( ,   identifier  "->" partition)* "end" ";" ;
partition ::= ( identifier ,  nat ,  nat ,  nat ) ;
```

Allocation des variables sources

```
AllocS ::= "allocS" ( identifier  "->"  identifier )
( ,   identifier  "->"  identifier )* "end" ";" ;
```

Allocation des variables puits

```
AllocP ::= "allocP" ( identifier  "->"  identifier )
( ,   identifier  "->"  identifier )* "end" ";" ;
```

Lien virtuel

Pour faciliter l'écriture du fichier d'entrée, la déclaration des liens virtuels se fait en amont de la déclaration de la fonction d'allocation des dépendances.

```
VL ::= "VL"  identifier  smin smax paths bag "end" ";" ;
smin ::= "smin"  nat   uniteTps  ";" ;
smax ::= "smax"  nat   uniteTps  ";" ;
paths ::= path ("," path)* ";" ;
path ::= "{"  identifier  ("," identifer )*"}" ;
bag ::= "bag"  nat   uniteTps  ";" ;
```

Exigences

Préalablement à la déclaration des exigences, nous devons déclarer l'ensemble des chaînes fonctionnelles.

```
requirements ::= (CF)+ (requirement)+ ;
```

Chaîne Fonctionnelle

```
CF ::= "CF"  identifier  ":"  identifier  "->"
( identifier  "->")*  identifier  "end" ";" ;
```

Exigence

Une exigence est déclarée avec son type (latence, fraîcheur, cohérence divergente ou convergente), la ou les chaîne(s) fonctionnelle(s) impliquée(s) et enfin la borne devant être respectée avec son sens , c'est-à-dire est-ce que la propriété doit être supérieure ou inférieure à cette borne.

```
requirement ::= "requirement"  typeExigence  chaines
 sensExigence   nat   uniteTps  "end" ";" ;
chaines ::= "("  identifier  ("," identifier )* ")" ;
```

Terminaux

```
// Terminaux
nat ::= ( 1 .. 9 )( 0 .. 9 )* |  0  ;
uniteTps ::= "ms" | "us" ;
uniteData ::= "b" | "kb" | "Mb" | "Gb" ;
uniteDebit ::= "b/s" | "kb/s" | "Mb/s" | "Gb/s" ;
typeNature ::= "periodic" | "sporadic" ;
typeExigence ::= "latency" | "freshness" | "consistD" | "consistC" ;
sensExigence ::= "<=" | ">=" ;


// Symboles ignorés
ignored ::=  cstylecomment |  spaces | newlines  ;
```

Application à l'étude de cas

Dans la suite, nous donnons un exemple de déclaration pour chaque élément défini dans la syntaxe du fichier d'entrée du prototype. Ces exemples sont tirés de l'étude de cas. A noter que la déclaration d'un port s'effectue conjointement à la déclaration de l'équipement auquel il est rattaché. Il n'est donc pas nécessaire de faire une déclaration spécifique pour les ports.

Architecture fonctionnelle

Variable

La déclaration du variable $speed_1$ se fait comme suit :

```
var speed1
nat periodic ;
size 0.8 kb ;
end ;
```

Fonction

La déclaration du FM1 se fait comme suit :

```
func FM1
periode 60 ms ;
WCET 30 ms ;
input wpId1, wpId2, answer1, speed1, speed2 ;
output wpInfo1, ETA1 ;
end ;
```

Dependance

Nous donnons la déclaration de la dépendance de la variable $pres_1$ avec l'environnement extérieur, c'est-à-dire (, req), ainsi que la dépendance ($req, speed_1$).

```
dep (varExt, pres1) end ;
dep (pres1, speed1) end ;
```

Architecture matérielle

Module

La déclaration du module M_1 se fait comme suit :

```
module M1
port M1.1 debit 100 Mb/s; ;
end;
```

Commutateur

La déclaration du commutateur S_1 se fait comme suit (seul les ports utilisés dans l'étude de cas sont déclarés) :

```
switch S1
lag 140 us ;
port S1.1 debit 100 Mb/s; ;
port S1.2 debit 100 Mb/s; ;
port S1.3 debit 100 Mb/s; ;
port S1.4 debit 100 Mb/s; ;
port S1.5 debit 100 Mb/s; ;
port S1.6 debit 100 Mb/s; ;
port S1.7 debit 100 Mb/s; ;
end ;
```

Lien

La déclaration du commutateur $l_1 = (S_1.1, M_1.1)$ reliant le module M_1 au commutateur S_1 se fait comme suit :

```
link l1 (S1.1,M1.1) end ;
```

Remote Data Concentrator

Comme vu dans la présentation du format d'entrée du prototype (c.f. sous-section 7.3.2 page 121), la déclaration du RDC R_1 est telle que :

```
RDC R1
periode 50 ms ;
WCET 10 ms ;
port R1.1 debit 100 Mb/s ; ;
end ;
```

Capteur

La déclaration du capteur du clavier du pilote key_1 se fait comme suit :

```
sensor key1
nat sporadic ;
periode 10 ms ;
TT [0.1, 0.2] ms ;
M1 ;
end ;
```

Actionneur

La déclaration de l'actionneur de l'écran du pilote $display_1$ se fait comme suit :

```
actuator display1
TT [0.1, 0.2] ms ;
M1 ;
end ;
```

Allocation

Allocation des fonctions

La déclaration de l'allocation des fonctions se fait comme suit (pour l'exemple, nous ne déclarons que les trois premières fonctions de la fonction d'allocation présentée dans l'exemple 4 (page 69)) :

```
AllocF
KU1 -> (M1,50,0,25),
MFD1 -> (M1,50,25,25),
KU2 -> (M2,50,0,25)
end ;
```

Allocation des variables sources

La déclaration de l'allocation des variables sources se fait comme suit :

```
AllocS
req1 -> key1,
req2 -> key2,
pres1 -> sensor1,
pres2 -> sensor2
end ;
```

Allocation des variables puits

La déclaration de l'allocation des variables puits se fait comme suit :

```
AllocS
disp1 -> display1,
disp2 -> display2
end ;
```

Lien virtuel

La déclaration du lien virtuel VL_1 se fait comme suit :

```
VL vl1
smin 600 b ;
smax 600 b ;
paths {l1,l2,l3}, {l1,l4,l5};
bag 32 ms;
end ;
```

Allocation des dépendances

La déclaration de l'allocation des dépendances se fait comme suit (pour l'exemple, nous ne déclarons que les trois premières dépendances de la fonction d'allocation présentée dans l'exemple 7 (page 73)) :

```
AllocD
(wpId, query1) -> vl1,
(wpId, query2) -> vl1,
(wpInfo1, disp1) -> vl3
end ;
```

Exigences

Chaine fonctionnelle

Nous déclarons les deux chaînes fonctionnelles $\mathcal{F}_1$ et $\mathcal{F}_2$ impliquées dans l'exigence de cohérence sur chaînes convergentes $ConsistC_{\leq 300}(\mathcal{F}_1, \mathcal{F}_2)$.

```
CF F1 : pres1 -> ADIRU1 -> speed1 -> FM1 -> ETA1 -> MFD1 -> disp1 end ;
CF F2 : pres2 -> ADIRU2 -> speed2 -> FM1 -> ETA1 -> MFD1 -> disp1 end ;
```

Exigence

Nous déclarons l'exigence $ConsistC_{\leq 300}(\mathcal{F}_1, \mathcal{F}_2)$.

```
requirement consistC (F1, F2) <= 300 ms end ;
```

Annexe B

Programmes linéaires des exigences de l étude de cas

Dans cette annexe sont donnés les programmes linéaires permettant la vérification des exigences de l'étude de cas.

Exigences de latence

Pour rappel, les deux exigences de latence à vérifier sur l'étude de cas sont :

$$Latency_{\leq 700}(\mathcal{L}_1) \text{ et } Latency_{\leq 700}(\mathcal{L}_2)$$

avec les chaînes fonctionnelles $\mathcal{L}_1$ et $\mathcal{L}_2$ telles que :

$$\mathcal{L}_1 = {}^{req}\, KU_1\, {}^{wpId}\, FM_1\, {}^{query_1}\, NDB\, {}^{answer_1}\, FM_1\, {}^{wpInfo_1}\, MFD_1\, {}^{disp_1}$$
$$\mathcal{L}_2 = {}^{req}\, KU_1\, {}^{wpId}\, FM_2\, {}^{query_2}\, NDB\, {}^{answer_2}\, FM_2\, {}^{wpInfo_2}\, MFD_2\, {}^{disp_2}$$

Nous détaillons le programme linéaire permettant de déterminer la latence pire cas de la chaîne $\mathcal{L}_1$, c'est-à-dire $WCL(\mathcal{L}_1)$.

Fonction objectif. L'indice du premier événement choisi en entrée de la chaîne est i_1 et l'estampille de l'événement est $t_{i_1}^{req_key_1}$. L'estampille du dernier événement est $t_{i_8}^{disp_1_display_1}$, ce qui donne la fonction objectif :

$$\max : t_{i_8}^{disp_1_display_1} - t_{i_1}^{req_key_1}$$

Système de contraintes.

$$\begin{aligned}
&// \ i_2^{\text{ième}} \text{ activation de } KU_1 \text{ est la première traitant } s_{req_key_1}[i_1] \\
&t_{i_1}^{req_key_1} \leq t_{i_2}^{KU_1} \\
&t_{i_2-1}^{KU_1} < t_{i_1}^{req_key_1} \\
&t_{i_2-1}^{KU_1} = t_{i_2}^{KU_1} - T_{KU_1}
\end{aligned}$$

$$\begin{aligned}
&// \text{ production de } wpId \\
&t_{i_2}^{KU_1} \leq t_{i_2}^{wpId_KU_1} \\
&t_{i_2}^{wpId_KU_1} \leq t_{i_2}^{KU_1} + C_{KU_1}
\end{aligned}$$

// traversée du lisseur de trafic de vl_1

$$t_{i_2}^{wpId_vl_1} = t_{i_2}^{wpId_KU_1} + n_{vl_1} \cdot vl_1.bag$$

// traversée du canal temporisé C_1

$$t_{i_2}^{wpId_vl_1} + a_1 \leq t_{i_2}^{wpId_C_1}$$

$$t_{i_2}^{wpId_C_1} \leq t_{i_2}^{wpId_vl_1} + b_1$$

// $i_3^{\text{ième}}$ activation de FM_1 est la première traitant $s_{wpId_C_1}[i_2]$

$$t_{i_2}^{wpId_C_1} \leq t_{i_3}^{FM_1}$$

$$t_{i_3-1}^{FM_1} < t_{i_2}^{wpId_C_1}$$

$$t_{i_3-1}^{FM_1} = t_{i_3}^{KU_1} - T_{FM_1}$$

// production de $query_1$

$$t_{i_3}^{FM_1} \leq t_{i_3}^{query_1_FM_1}$$

$$t_{i_3}^{query_1_FM_1} \leq t_{i_3}^{FM_1} + C_{FM_1}$$

// traversée du lisseur de trafic de vl_4

$$t_{i_3}^{query_1_vl_4} = t_{i_3}^{query_1_FM_1} + n_{vl_4} \cdot vl_4.bag$$

// traversée du canal temporisé C_4

$$t_{i_3}^{query_1_vl_4} + a_4 \leq t_{i_3}^{query_1_C_4}$$

$$t_{i_3}^{query_1_C_4} \leq t_{i_3}^{query_1_vl_4} + b_4$$

// $i_4^{\text{ième}}$ activation de NDB est la première traitant $s_{query_1_C_4}[i_3]$

$$t_{i_3}^{query_1_C_4} \leq t_{i_4}^{NDB}$$

$$t_{i_4-1}^{NDB} < t_{i_3}^{query_1_C_4}$$

$$t_{i_4-1}^{NDB} = t_{i_4}^{NDB} - T_{NDB}$$

// production de $answer_1$

$$t_{i_4}^{NDB} \leq t_{i_4}^{query_1_NDB}$$

$$t_{i_4}^{answer_1_NDB} \leq t_{i_4}^{NDB} + C_{NDB}$$

// traversée du lisseur de trafic de vl_7

$$t_{i_4}^{answer_1_vl_7} = t_{i_4}^{answer_1_NDB} + n_{vl_7} \cdot vl_7.bag$$

// traversée du canal temporisé C_7

$$t_{i_4}^{answer_1_vl_7} + a_7 \leq t_{i_4}^{answer_1_C_7}$$

$$t_{i_4}^{answer_1_C_7} \leq t_{i_4}^{answer_1_vl_7} + b_7$$

// i_5 ième activation de FM_1 est la première traitant $s_{answer_1_C_7}[i_4]$

$t_{i_4}^{answer_1_C_7} \leq t_{i_5}^{FM_1}$

$t_{i_5-1}^{FM_1} < t_{i_4}^{answer_1_C_7}$

$t_{i_5-1}^{FM_1} = t_{i_5}^{FM_1} - T_{FM_1}$

// production de $wpInfo_1$

$t_{i_5}^{FM_1} \leq t_{i_5}^{wpInfo_1_FM_1}$

$t_{i_5}^{wpInfo_1_FM_1} \leq t_{i_5}^{FM_1} + C_{FM_1}$

// traversée du lisseur de trafic de vl_3

$t_{i_5}^{wpInfo_1_vl_3} = t_{i_5}^{wpInfo_1_FM_1} + n_{vl_3} \cdot vl_3.bag$

// traversée du canal temporisé C_3

$t_{i_5}^{wpInfo_1_vl_3} + a_3 \leq t_{i_5}^{wpInfo_1_C_3}$

$t_{i_5}^{wpInfo_1_C_3} \leq t_{i_5}^{wpInfo_1_vl_3} + b_3$

// i_6 ième activation de MFD_1 est la première traitant $s_{wpInfo_1_C_3}[i_5]$

$t_{i_5}^{answer_1_C_3} \leq t_{i_6}^{MFD_1}$

$t_{i_6-1}^{MFD_1} < t_{i_5}^{wpInfo_1_C_3}$

$t_{i_6-1}^{MFD_1} = t_{i_6}^{MFD_1} - T_{MFD_1}$

// production de $disp_1$

$t_{i_6}^{MFD_1} \leq t_{i_6}^{disp_1_MFD_1}$

$t_{i_6}^{disp_1_MFD_1} \leq t_{i_6}^{MFD_1} + C_{MFD_1}$

// affichage $disp_1$ l'par actionneur $display_1$

$t_{i_6}^{disp_1_MFD_1} + a_{display_1} \leq t_{i_6}^{disp_1_display_1}$

$t_{i_6}^{disp_1_display_1} \leq t_{i_6}^{disp_1_display_1} + b_{display_1}$

// activations des fonctions et des capteurs

$t_{i_1}^{req_key_1} = \phi_{key_1} + i_1 \cdot T_{key_1} + \delta_{key_1}$

$t_{i_2}^{KU_1} = \phi_{M_1} + O_{KU_1} + i_2 \cdot T_{KU_1}$

$t_{i_6}^{MFD_1} = \phi_{M_1} + O_{MFD_1} + i_6 \cdot T_{MFD_1}$

$t_{i_3}^{FM_1} = \phi_{M_3} + O_{FM_1} + i_3 \cdot T_{FM_1}$

$t_{i_5}^{FM_1} = \phi_{M_3} + O_{FM_1} + i_5 \cdot T_{FM_1}$

$t_{i_4}^{NDB} = \phi_{M_7} + O_{NDB} + i_4 \cdot T_{NDB}$

Domaine des variables.

$$
\begin{array}{l}
// \text{ phases des modules et des capteurs} \\
\phi_{key_1} \quad [0, T_{key_1}[\\
\phi_{M_1} \quad [0, HP(M_1)[\\
\phi_{M_3} \quad [0, HP(M_3)[\\
\phi_{M_7} \quad [0, HP(M_7)[\\
n_{vl_1} = 0..N_{vl_1} - 1 \\
n_{vl_4} = 0..N_{vl_4} - 1 \\
n_{vl_3} = 0..N_{vl_3} - 1 \\
(i_1, i_2, i_3, i_4, i_5, i_6) \quad \mathbb{N}^6 \\
t_{i_1}^{req_key_1} \quad \mathbb{R}^+ \\
(t_{i_2}^{KU_1}, t_{i_2-1}^{KU_1}, t_{i_2}^{wpId_KU_1}, t_{i_2}^{wpId_vl_1}, t_{i_2}^{wpId_C_1}) \quad (\mathbb{R}^+)^5 \\
(t_{i_3}^{FM_1}, t_{i_3-1}^{FM_1}, t_{i_3}^{query_1_FM_1}, t_{i_3}^{query_1_vl_4}, t_{i_3}^{query_1_C_4}) \quad (\mathbb{R}^+)^5 \\
(t_{i_4}^{NDB}, t_{i_4-1}^{NDB}, t_{i_4}^{answer_1_NDB}, t_{i_4}^{answer_1_vl_7}, t_{i_4}^{answer_1_C_7}) \quad (\mathbb{R}^+)^5 \\
(t_{i_5}^{FM_1}, t_{i_5-1}^{FM_1}, t_{i_5}^{wpInfo_1_FM_1}, t_{i_5}^{wpInfo_1_vl_3}, t_{i_5}^{wpInfo_1_C_3}) \quad (\mathbb{R}^+)^5 \\
(t_{i_6}^{MFD_1}, t_{i_6+1}^{MFD_1}, t_{i_6}^{disp_1_MFD_1}, t_{i_6}^{disp_1_display_1}) \quad (\mathbb{R}^+)^5
\end{array}
$$

Exigence de cohérence entre chaînes divergentes

Pour rappel, l'exigence de cohérence entre chaînes divergentes à vérifier sur l'étude de cas est :

$$ConsistD_{\leq 500}(\mathcal{L}_1, \mathcal{L}_2) \tag{B.1}$$

avec les chaînes fonctionnelles $\mathcal{L}_1$ et $\mathcal{L}_2$ telles que présentées pour les exigences de latence.

Fonction objectif. La fonction objectif utilise deux variables intermédiaires $maxT$ et $minT$

$$\max : maxT - minT$$

Contraintes spéci ques à la fonction objectif. Les variables $maxT$ et $minT$ sont contraintes par les estampilles des événements considérés à la sortie des chaînes fonctionnelles : $t_{i_6}^{disp_1_display_1}$ et $t_{i'_6}^{disp_2_display_2}$.

$$
\begin{cases}
maxT \leq t_{i_6}^{disp_1_display_1} + b_{disp_1}^{maxT} \cdot M \\
maxT \geq t_{i_6}^{disp_1_display_1} - b_{disp_1}^{maxT} \cdot M \\
maxT \leq t_{i'_6}^{disp_2_display_2} + b_{disp_2}^{maxT} \cdot M \\
maxT \geq t_{i'_6}^{disp_2_display_2} - b_{disp_2}^{maxT} \cdot M \\
b_{disp_1}^{maxT} + b_{disp_2}^{maxT} = 1 \\
minT \leq t_{i_6}^{disp_1_display_1} + b_{disp_1}^{minT} \cdot M \\
minT \geq t_{i_6}^{disp_1_display_1} - b_{disp_1}^{minT} \cdot M \\
minT \leq t_{i'_6}^{disp_2_display_2} + b_{disp_2}^{minT} \cdot M \\
minT \geq t_{i'_6}^{disp_2_display_2} - b_{disp_2}^{minT} \cdot M \\
b_{disp_1}^{minT} + b_{disp_2}^{minT} = 1
\end{cases}
$$

$b_{disp_1}^{maxT}$, $b_{disp_2}^{maxT}$, $b_{disp_1}^{minT}$ et $b_{disp_2}^{minT}$ sont des variables booléennes.

Annexe C

Méthode analytique locale

Dans cette annexe nous présentons les formules analytiques *locales*, que nous utilisons pour comparer les résultats de la méthode de vérification proposée dans cette thèse. Ces formules sont obtenues en considérant les pires comportements des éléments du système. Le pire cas d'utilisation d'un élément est déterminé localement, c'est-à-dire sans considérer le comportement global du système, comme cela est fait avec la méthode par programmation linéaire vue précédemment.

Pire et meilleur temps de traversée des éléments du système abstrait

Dans un premier temps, nous listons pour chaque élément du système abstrait ses pire et meilleur cas d'utilisation. Ceci nous permettra par la suite de déterminer des formules analytiques *locales*, donnant des bornes supérieures ou inférieures aux exigences.

Capteur. Une occurrence de variable est retardée dans un capteur S, quelle que soit sa nature, par le temps de traversée du bus le reliant à l'équipement auquel il est rattaché. Ce temps de traversée est inclus dans un intervalle $[a_S, b_S]$. Le pire temps de traversée du capteur est alors b_S et le meilleur temps a_S.

RDC. Pour un RDC r, de période T_r et de temps de traitement maximal C_r, le pire cas de traversée pour une occurrence de variable correspond à l'arrivée de l'occurrence juste après l'activation du RDC. Cette occurrence doit alors attendre la prochaine activation du RDC, soit un temps T_r, avant d'être traitée. Dans le pire cas, ce traitement prendra alors un temps C_r. Le pire temps de traversée d'un RDC r est alors : $T_r + C_r$.

Le meilleur cas de traversée correspond à l'arrivée de l'occurrence à l'instant d'activation du RDC. L'occurrence ne subit donc pas de temps d'attente avant le début du traitement et dans le meilleur cas, ce traitement est instantané. Le meilleur temps de traversée d'un RDC r est alors égal à 0.
Remarque : il n'y a pas de difficulté technique à prendre en compte un temps de traitement minimum dans le RDC ou dans une fonction. Nous ne l'avons pas fait pour ne pas alourdir le modèle.

Fonction. Pour une fonction f, de période T_f et de temps de traitement maximal C_f, le pire cas de traversée pour une occurrence de variable correspond à l'arrivée de l'occurrence juste après l'activation de la fonction. Cette occurrence doit alors attendre la prochaine activation de la fonction, soit un temps T_f, avant d'être traitée. Dans le pire cas, ce traitement prendra alors un temps C_f. Le pire temps de traversée d'une fonction f est alors : $T_f + C_f$.

Le meilleur cas de traversée correspond à l'arrivée de l'occurrence à l'instant d'activation de la fonction. L'occurrence ne subit donc pas de temps d'attente avant le début du traitement et dans le meilleur cas, ce traitement est instantané. Le meilleur temps de traversée d'une fonction est alors égal à 0.

Lisseur de tra c. Soit un lisseur de trafic *vl*, avec comme paramètres : *bag*, le temps minimum d'émission entre chaque occurrence et N_{vl} le nombre maximum d'occurrences pouvant arriver simultanément.

Le pire cas pour une occurrence correspond au cas où le lisseur de trafic a reçu N_{VL} occurrences à traiter et que l'occurrence est traitée en dernier. Le pire temps de traversée du lisseur de trafic est alors : $(N_{vl} - 1) \cdot bag$.

Le meilleur cas se produit quand l'occurrence est traitée en premier par le lisseur de trafic, peu importe le nombre d'occurrences reçues. La transmission de l'occurrence est alors instantanée. Le meilleur temps de traversée d'un lisseur de trafic est alors égal à 0.

Canal temporisé. Pour un canal temporisé C dont le temps de traversée est borné par l'intervalle $[a_C, b_C]$. Le pire temps est donc b_C, et le meilleur temps est a_C.

Actionneur. Une occurrence de variable est retardée dans un actionneur A par le temps de traversée du bus le reliant à l'équipement auquel il est rattaché. Ce temps de traversée est inclus dans un intervalle $[a_A, b_A]$. Le pire temps de traversée du capteur est alors b_A, et le meilleur temps a_A.

Ces pires et meilleurs temps de traversée sont résumés dans le tableau C.1.

	Pire temps	Meilleur temps
Capteur S	b_S	a_S
RDC r	$T_r + C_r$	0
Fonction f	$T_f + C_f$	0
Lisseur de trafic vl	$(N_{vl} - 1) \cdot bag$	0
Canal temporisé C	b_C	a_C
Actionneur A	b_A	a_A

Table C.1 – Pires et meilleurs temps de traversée des éléments du système abstrait

Latence

Une borne supérieure de la latence pire cas d'une chaîne fonctionnelle est alors donnée par la somme des pires temps de traversée de chacun des éléments traversés par la chaîne. Une borne supérieure de la latence meilleur cas est donnée par la somme des meilleurs temps de traversée de ces éléments.

Nous donnons les formules pour une chaîne fonctionnelle générique $\mathcal{F} =^{v_0} F_1{}^{v_1} \ldots F_n{}^{v_n}$. La variable v_0 est produite par un capteur S, rattaché à un RDC r_S. La variable v_n est utilisée par un actionneur A, rattaché à un RDC r_A. Chaque variable v_i, $i = 0..n$, traverse le réseau via un lisseur de trafic VL_i et un canal temporisé C_k. Les caractéristiques temporelles du capteur S, de l'actionneur A et des canaux C_k sont respectivement les intervalles $[a_S, b_S], [a_A, b_A]$ et $[a_{C_k}, b_{C_k}]$. Une borne supérieure de la latence pire cas, notée $\overline{WCL}$, est alors :

$$\begin{aligned}\overline{WCL}(\mathcal{F}) =& b_S + T_{r_S} + C_{r_S} + (N_{VL_0} - 1) \cdot bag_{VL_0} + b_{C_0} \\ &+ \sum_{i=1}^{n} \left(T_{F_i} + C_{F_i} + (N_{VL_i} - 1) \cdot bag_{VL_i} + b_{C_i}\right) \\ &+ T_{r_A} + C_{r_1} + b_A\end{aligned} \tag{C.1}$$

Pour le pire cas, la formule est similaire à celle utilisée par les auteurs de [6]. Une borne inférieure de la latence meilleur cas, notée $\underline{BCL}$, est donnée par :

$$\underline{BCL}(\mathcal{F}) = a_S + \sum_{i=0}^{n} a_{C_i} + a_A \tag{C.2}$$

Prenons un exemple tiré de l'étude de cas : la chaîne $\mathcal{L}_1 =^{req_1} KU_1{}^{wpId_1} FM_1{}^{query_1} NDB{}^{answer_1} FM_1{}^{wpInfo_1} MFD_1{}^{disp_1}$. Cette chaîne traverse les éléments suivants : le capteur key_1, la fonction KU_1, le lisseur de trafic VL_1, le canal temporisé C_1, la fonction FM_1, le lisseur de trafic VL_4, le canal temporisé C_4, la fonction NDB, le lisseur de trafic VL_7, le canal temporisé C_7, la fonction FM_1, le lisseur de trafic

VL_3, le canal temporisé C_3, la fonction MFD_1 et finalement l'actionneur $display_1$. Une borne supérieure de la latence pire cas de la chaîne $\mathcal{L}_1$ est alors :

$$\begin{aligned}\overline{WCL}(\mathcal{L}_1) = b_{key_1} &+ T_{KU_1} + C_{KU_1} + (N_{VL_1} - 1) \cdot bag_{VL_1} + b_{C_1} \\ &+ T_{FM_1} + C_{FM_1} + (N_{VL_4} - 1) \cdot bag_{VL_4} + b_{C_4} \\ &+ T_{NDB} + C_{NDB} + (N_{VL_7} - 1) \cdot bag_{VL_7} + b_{C_7} \\ &+ T_{FM_1} + C_{FM_1} + (N_{VL_3} - 1) \cdot bag_{VL_3} + b_{C_3} \\ &+ T_{MFD_1} + C_{MFD_1} + b_{display_1} = 524,132ms\end{aligned} \tag{C.3}$$

Une borne inférieure de la latence meilleur cas de la chaîne $\mathcal{L}_1$ est alors :

$$\begin{aligned}\underline{BCL}(\mathcal{L}_1) &= a_{key_1} + a_{C_1} + a_{C_4} + a_{C_7} + a_{C_3} + a_{display_1} \\ &= 1,358ms\end{aligned} \tag{C.4}$$

Fraîcheur

La définition d'une borne supérieure pour la fraîcheur, notée $\overline{WCF}$, est moins intuitive que celle de la latence. Soit une chaîne fonctionnelle $\mathcal{F} = {}^{v_0}F_1{}^{v_1} \ldots F_n{}^{v_n}$. La variable v_0 est produite par un capteur S. La variable v_n est utilisée par un actionneur A. Chaque variable v_i, $i = 0..n$, traverse le réseau via un lisseur de trafic VL_i et un canal temporisé C_k. Les caractéristiques temporelles du capteur S, de l'actionneur A et des canaux C_k sont respectivement les intervalles $[a_S, b_S]$, $[a_A, b_A]$ et $[a_{C_k}, b_{C_k}]$. Pour ne pas compliquer la présentation, nous n'utilisons pas de RDC pour le capteur et l'actionneur. Leur prise en compte dans la borne de la fraîcheur est similaire à la prise en compte des fonctions.

Considérons une exécution le long de la chaîne fonctionnelle telle que la ${i_A}^{\text{ième}}$ occurrence de la variable v_n en sortie soit la dernière occurrence à dépendre de la ${i_S}^{\text{ième}}$ occurrence de la variable v_0 en entrée. Nous avons donc $s_{v_0_S}[i_S] \quad s_{v_n_A}[i_n]$ et la distance entre ces deux événements représente la pire fraîcheur atteignable à partir de cette occurrence en entrée. Cette fraîcheur s'exprime :

$$F = t_{i_A}^{v_n_A} - t_{i_S}^{v_0_S} \tag{C.5}$$

Nous cherchons alors une borne supérieure à cette fraîcheur. En effet, cette paire d'événements étant choisie en toute généralité, cette borne pourra être utilisée pour définir une borne supérieure pour la fraîcheur de la chaîne.

Supposons que pour tout k, la ${i_k}^{\text{ième}}$ occurrence de la variable v_k est la dernière occurrence dépendante de la ${i_{k-1}}^{\text{ième}}$ occurrence de la variable v_{k-1}. Ces notations sont introduites sur la figure C.1.

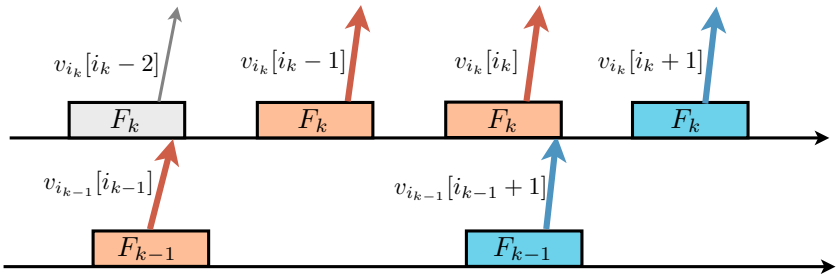


FIGURE C.1 – Notation des indices d'occurrences utilisés

Nous pouvons alors réécrire F de la façon suivante :

$$F = \underbrace{t_{i_A}^{v_n_A} - t_{i_n}^{v_n_F_n}}_{(A)} + \underbrace{t_{i_n}^{v_n_F_n} - t_{i_n}^{F_n}}_{(B)} + \underbrace{t_{i_n}^{F_n} - t_{i_1}^{F_1}}_{(C)} + \underbrace{t_{i_1}^{F_1} - t_{i_S}^{v_0_S}}_{(D)} \tag{C.6}$$

Dans la suite, nous bornons chacun des termes A, B, C et D. Le terme $A = t_{i_A}^{v_n_A} - t_{i_n}^{v_n_F_n}$ représente le temps écoulé entre la production de l'occurrence en sortie de F_n et son utilisation dans l'actionneur.

En utilisant le pire cas d'utilisation défini pour la latence, ce temps est borné par b_A, ainsi :

$$A \leq b_A \tag{C.7}$$

Le terme $B = t_{i_n}^{v_n_F_n} - t_{i_n}^{F_n}$ représente le temps d'exécution de la fonction F_n. Celui-ci est borné par C_{F_n}, ainsi :

$$B \leq C_{F_n} \tag{C.8}$$

Nous cherchons désormais une borne pour le terme $C = t_{i_n}^{v_n_F_n} - t_{i_1}^{v_1_F_1}$. Si la $i_{k-1}{}^{\text{ième}}$ occurrence de la variable v_{k-1} est la dernière dépendant de la $i_{k-1}{}^{\text{ième}}$ occurrence de la variable v_{k-1}, cela signifie que la $i_{k-1} + 1^{\text{ième}}$ occurrence de la variable v_{k-1} arrive nécessairement après la $i_k{}^{\text{ième}}$ activation de F_k. L'estampille de l'événement représentant la sortie du canal temporisée C_{k-1} de cette occurrence est $t_{i_{k-1}+1}^{v_{k-1}_C_{k-1}}$. Nous avons donc la relation :

$$t_{i_k}^{F_k} < t_{i_{k-1}+1}^{v_{k-1}_C_{k-1}} \tag{C.9}$$

L'instant d'arrivée $t_{i_{k-1}+1}^{v_{k-1}_C_{k-1}}$ est borné par la durée maximale d'exécution de F_{k-1} et le délai de communication maximal entre F_{k-1} et F_k. En utilisant les pires cas d'utilisation des lisseurs de trafic et des canaux temporisés introduits pour la latence, nous avons :

$$t_{i_{k-1}+1}^{v_{k-1}_C_{k-1}} \leq t_{i_{k-1}+1}^{F_{k-1}} + C_{F_{k-1}} + b_{C_{k-1}} + (N_{VL_{k-1}} - 1) \cdot BAG_{VL_{k-1}}$$

Par définition, $t_{i_{k-1}+1}^{F_{k-1}} = t_{i_{k-1}}^{F_{k-1}} + T_{F_{k-1}}$, ainsi :

$$t_{i_{k-1}+1}^{v_{k-1}_C_{k-1}} \leq t_{i_{k-1}}^{F_{k-1}} + T_{F_{k-1}} + C_{F_{k-1}} + b_{C_{k-1}} + (N_{VL_{k-1}} - 1) \cdot BAG_{VL_{k-1}}$$

Avec l'inégalité (C.9), nous obtenons :

$$\begin{array}{lll} t_{i_n}^{F_n} & \leq t_{i_{n-1}}^{F_{n-1}} & +T_{F_{n-1}} + C_{F_{n-1}} + b_{C_{n-1}} + (N_{VL_{n-1}} - 1) \cdot BAG_{VL_{n-1}} \\ & \leq t_{i_{n-2}}^{F_{n-2}} & +T_{F_{n-2}} + C_{F_{n-2}} + b_{C_{n-2}} + (N_{VL_{n-2}} - 1) \cdot BAG_{VL_{n-2}} \\ & & +T_{F_{n-1}} + C_{F_{n-1}} + b_{C_{n-1}} + (N_{VL_{n-1}} - 1) \cdot BAG_{VL_{n-1}} \\ & \vdots & \\ t_{i_n}^{F_n} & \leq t_{i_1}^{F_1} & +\sum_{k=1}^{n-1} [T_{F_k} + C_{F_k} + b_{C_k} + (N_{VL_k} - 1) \cdot BAG_{VL_k}] \end{array} \tag{C.10}$$

On en déduit une borne pour le terme $C = t_{i_n}^{v_n_F_n} - t_{i_1}^{v_1_F_1}$:

$$C \leq \sum_{k=1}^{n-1} [T_{F_k} + C_{F_k} + b_{C_k} + (N_{VL_k} - 1) \cdot BAG_{VL_k}] \tag{C.11}$$

Le terme $D = t_{i_1}^{F_1} - t_{i_S}^{v_0_S}$ représente le temps séparant la $i_S{}^{\text{ième}}$ mesure du capteur à l'activation de F_1 l'utilisant pour la dernière fois. Ce temps ne peut pas être supérieur au temps maximum séparant deux mesures consécutives du capteur, soit $T_S - (b_S - a_S)$, ainsi :

$$D \leq T_S - (b_S - a_S) \tag{C.12}$$

Des inégalités (C.7), (C.8), (C.11) et (C.7) nous pouvons déduire une borne supérieure de F, nous utilisons cette borne pour définir une borne supérieure à la fraîcheur pire cas d'une chaîne fonctionnelle $\mathcal{F}$:

$$\overline{WCF}(F) = b_A + C_{F_n} + \sum_{k=1}^{n-1} [T_{F_k} + C_{F_k} + b_{C_k} + (N_{VL_k} - 1) \cdot BAG_{VL_k}] + T_S - (b_S - a_S) \tag{C.13}$$

Comme remarqué lors de la formalisation des exigences (c.f. chapitre 6 page 97), la fraîcheur meilleur cas est équivalente à la latence meilleur cas. Nous utiliserons donc la borne inférieure de la latence meilleur cas.

En appliquant ces formules à la chaîne $\mathcal{F}_1$ de l'étude de cas, nous obtenons :

$$\begin{aligned}\overline{WCF}(\mathcal{F}_1) = b_{display_1} &+ C_{MFD_1} \\ &+ T_{FM_1} + C_{FM_1} + (N_{VL_3} - 1) \cdot bag_{VL_3} + b_{C_3} \\ &+ T_{ADIRU_1} + C_{ADIRU_1} + (N_{VL_1 1} - 1) \cdot bag_{VL_1 1} + b_{C_1 1} \\ &+ T_{RDC_1} + C_{RDC_1} + (N_{VL_9} - 1) \cdot bag_{VL_9} + b_{C_9} \\ &+ T_{sensor_1} - (b_{sensor_1} - a_{sensor_1}) = 294,48ms\end{aligned} \tag{C.14}$$

Une borne inférieure de la fraîcheur meilleur cas de la chaîne $\mathcal{F}_1$ est :

$$\begin{aligned}\underline{BCL}(\mathcal{F}_1) &= a_{sensor_1} + a_{C_9} + a_{C_1 1} + a_{C_4} + a_{display_1} \\ &= 1,052ms\end{aligned} \tag{C.15}$$

Cohérence entre chaînes divergentes

Nous évaluons une borne de la cohérence pire cas entre chaînes fonctionnelles divergentes en considérant que le pire cas survient quand une des chaînes subit son pire cas alors qu'une autre subit son meilleur cas. Cette approche est similaire à ce que proposent les auteurs de [90], bien que les systèmes considérés soient différents. Une borne supérieure de la cohérence est alors la plus grande distance entre les latences pire et meilleur cas des chaînes fonctionnelles. En notant $\overline{WCCD}$ cette borne, pour un ensemble de chaînes fonctionnelles $\mathcal{F}_1, \ldots, \mathcal{F}_n$, nous avons :

$$\overline{WCCD}(\mathcal{F}_1, \ldots, \mathcal{F}_n) = \max_{i=1..n} \left\{ \overline{WCL}(\mathcal{F}_i) \right\} - \min_{i=1..n} \ \underline{BCL}(\mathcal{F}_i) \tag{C.16}$$

A noter qu'il peut être nécessaire de considérer des sous-chaînes de $\mathcal{F}_1, \ldots, \mathcal{F}_n$ dans l'expression de $\overline{WCCD}$. En effet, par définition, des chaînes fonctionnelles divergentes partagent un début commun et donc elles ne perdent de la cohérence qu'à partir du moment où elles divergent. Prenons l'exemple de l'exigence de cohérence entre les chaînes $\mathcal{L}_1$ et $\mathcal{L}_2$ de l'étude de cas, représentée sur la figure C.2.

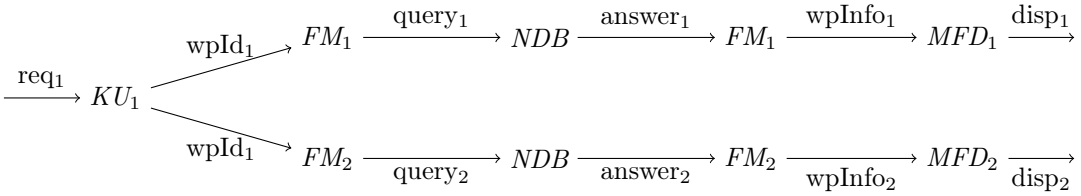


FIGURE C.2 – Chaînes divergentes de l'étude de cas

En sortie de KU_1, la variable $wpId_1$ est transmise à FM_1 et à FM_2 au travers du lisseur de trafic VL_1. En sortie du lisseur, la variable emprunte le canal temporisé C_1 pour rejoindre FM_1 et C_1 pour rejoindre FM_2. C'est à cet endroit que les chaînes fonctionnelles divergent et donc les pires et meilleurs latences utilisées ne doivent être mesurées qu'à partir de ces canaux temporisés et jusqu'au bout des chaînes.

Sur la figure C.3, nous représentons les intervalles $[\underline{BCL}(\mathcal{F}_i), \overline{WCL}(\mathcal{F}_i)]$ composés des bornes des latences meilleur et pire cas des chaînes fonctionnelles $\mathcal{F}_1$, $\mathcal{F}_2$ et $\mathcal{F}_3$.

D'une manière générale, on peut remarquer que le meilleur cas de cohérence n'est pas nul si les intervalles $[\underline{BCL}(\mathcal{F}_i), \overline{WCL}(\mathcal{F}_i)]$ sont disjoints deux à deux. Sur l'exemple, de la figure C.3, on a :

$$[\underline{BCL}(\mathcal{F}_1), \overline{WCL}(\mathcal{F}_1)] \setminus [\underline{BCL}(\mathcal{F}_2), \overline{WCL}(\mathcal{F}_2)] = [\underline{BCL}(\mathcal{F}_2), \overline{WCL}(\mathcal{F}_1)] = \tag{C.17}$$

La meilleure cohérence entre $\mathcal{F}_1$ et $\mathcal{F}_2$ est alors égale à 0. De la même manière, l'intersection des intervalles de $\mathcal{F}_1$ et $\mathcal{F}_3$ est non vide et leur meilleure cohérence est alors égale à 0. En revanche, entre $\mathcal{F}_2$

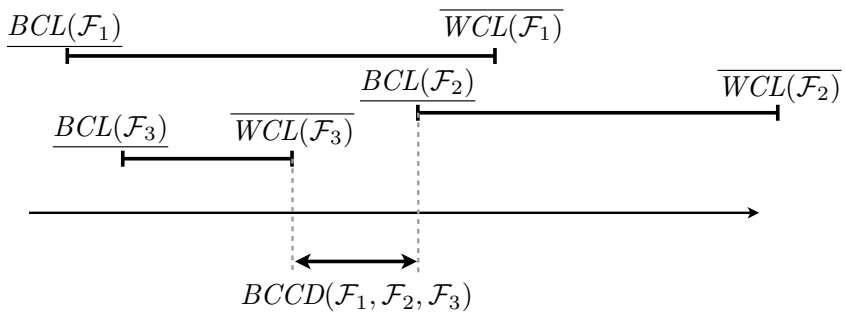


FIGURE C.3 – Illustration du meilleur cas de cohérence

et $\mathcal{F}_3$, l'intersection de leurs intervalles est vide, la meilleure cohérence est alors égale à la distance entre ces intervalles, soit :

$$\underline{BCCD}(\mathcal{F}_2, \mathcal{F}_3) = \underline{BCL}(\mathcal{F}_2) - \overline{WCL}(\mathcal{F}_3) \tag{C.18}$$

Le distance séparant deux intervalles $[a_1, b_1]$, $[a_2, b_2]$ peut s'exprimer :

$$\max\ a_1 - b_2; a_2 - b_1; 0 \tag{C.19}$$

On en déduit une formule donnant le meilleur cas de cohérence entre les chaînes divergentes $\mathcal{F}_1, \ldots, \mathcal{F}_n$:

$$\underline{BCCD}(\mathcal{F}_1, \ldots, \mathcal{F}_n) = \max_{\substack{i,j = 1..n \\ i = j}} \left\{ \underline{BCL}(\mathcal{F}_i) - \overline{WCL}(\mathcal{F}_i) \right\}^+ \tag{C.20}$$

avec $x \quad \mathbb{R}$, $x^+ = \max\ x; 0$.

Nous appliquons ces formules à l'étude de cas. Nous calculons tout d'abord les pires et meilleures latences. Nous notons l_1 et l_2 les chaînes correspondant aux chaînes $\mathcal{L}_1$ et $\mathcal{L}_2$, « raccourcies » de leur début commun. Nous avons :

$$\begin{aligned} \overline{WCL}(l_1) = b_{C_1} &+ T_{FM_1} + C_{FM_1} + (N_{VL_4} - 1) \cdot bag_{VL_4} + b_{C_4} \\ &+ T_{NDB} + C_{NDB} + (N_{VL_7} - 1) \cdot bag_{VL_7} + b_{C_7} \\ &+ T_{FM_1} + C_{FM_1} + (N_{VL_3} - 1) \cdot bag_{VL_3} + b_{C_3} \\ &+ T_{MFD_1} + C_{MFD_1} + b_{display_1} = 449,032ms \end{aligned} \tag{C.21}$$

et :

$$\begin{aligned} \underline{BCL}(l_1) &= a_{C_1} + a_{C_4} + a_{C_7} + a_{C_3} + a_{display_1} \\ &= 1,258ms \end{aligned} \tag{C.22}$$

Le système étant symétrique nous avons : $\overline{WCL}(l_1) = \overline{WCL}(l_2)$ et $\underline{BCL}(l_1) = \underline{BCL}(l_2)$. Le meilleur cas de cohérence est nul car les intervalles $[\overline{WCL}(l_1), \underline{BCL}(l_1)]$ et $[\overline{WCL}(l_2), \underline{BCL}(l_2)]$ sont identiques. Le pire cas est :

$$\overline{WCCD}(\mathcal{L}_1, \mathcal{L}_2) = \overline{WCL}(l_1) - \underline{BCL}(l_2) = 449,032 - 1,258 = 447,774ms \tag{C.23}$$

Cohérence entre chaînes convergentes

Le pire cas de cohérence entre chaînes convergentes survient lorsque l'une des chaînes subit sa pire fraîcheur, et qu'une autre subit sa meilleure latence. Une borne supérieure de la cohérence entre chaînes convergentes est alors la plus grande distance entre la fraîcheur pire cas et la latence meilleur cas

des chaînes fonctionnelles. En notant $\overline{WCCC}$ cette borne, pour un ensemble de chaînes fonctionnelles $\mathcal{F}_1, \ldots, \mathcal{F}_n$, nous avons :

$$\overline{WCCC}(\mathcal{F}_1, \ldots, \mathcal{F}_n) = \max_{i=1..n} \left\{ \overline{WCF}(\mathcal{F}_i) \right\} - \min_{i=1..n} \ \underline{BCL}(\mathcal{F}_i) \tag{C.24}$$

De façon similaire à la définition d'une borne de la meilleure cohérence entre chaînes fonctionnelles divergentes, nous définissons $\overline{BCCC}$, qui est une borne de la meilleure cohérence entre chaînes fonctionnelles convergentes. Pour un ensemble de chaînes fonctionnelles convergentes $\mathcal{F}_1, \ldots, \mathcal{F}_n)$, nous avons :

$$\underline{BCCC}(\mathcal{F}_1, \ldots, \mathcal{F}_n) = \max_{\substack{i,j=1..n \\ i=j}} \left\{ \underline{BCL(\mathcal{F}_i)} - \overline{WCF(\mathcal{F}_i)} \right\}^+ \tag{C.25}$$

avec $x \quad \mathbb{R}$, $x^+ = \max\ x; 0$.

Comme pour la cohérence entre chaînes divergentes, il peut être nécessaire de considérer des sous-chaînes de $\mathcal{F}_1, \ldots, \mathcal{F}_n$ dans l'expression de $\overline{WCCC}$ et $\underline{BCCC}$. En effet, par définition, des chaînes fonctionnelles convergentes partagent une fin commune et donc elles ne perdent plus de cohérence à partir du moment où elles convergent. Prenons l'exemple de l'exigence de cohérence entre les chaînes $\mathcal{F}_1$ et $\mathcal{F}_2$ de l'étude de cas, représentée sur la figure C.4.

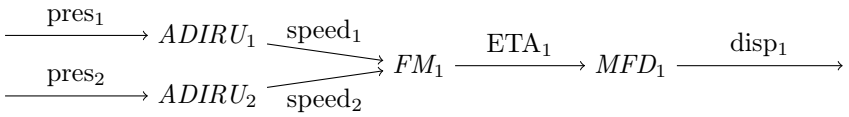


FIGURE C.4 – Chaînes convergentes de l'étude de cas

Les deux chaînes convergent en entrée de FM_1, c'est-à-dire à la sortie des canaux temporisés C_{11} et C_{12}. Les fraîcheurs pire cas, et les latences meilleur cas, utilisées doivent alors être mesurées entre le début des chaînes et la sortie de ces canaux.

Nous appliquons ces formules à l'étude de cas. Nous calculons tout d'abord les pires fraîcheurs et les meilleures latences. Nous notons f_1 et f_2 les chaînes correspondant aux chaînes $\mathcal{F}_1$ et $\mathcal{F}_2$, « raccourcies » de leur fin commune. Nous avons :

$$\begin{aligned} \overline{WCF}(f_1) = & T_{ADIRU_1} + C_{ADIRU_1} + (N_{VL_11} - 1) \cdot bag_{VL_{11}} + b_{C_{11}} \\ & + T_{RDC_1} + C_{RDC_1} + (N_{VL_9} - 1) \cdot bag_{VL_9} + b_{C_9} \\ & + T_{sensor_1} - (b_{sensor_1} - a_{sensor_1}) = 170,82ms \end{aligned} \tag{C.26}$$

et :

$$\begin{aligned} \underline{BCL}(f_1) &= a_{sensor_1} + a_{C_9} + a_{C_{11}} \\ &= 0,682ms \end{aligned} \tag{C.27}$$

Le système étant symétrique nous avons : $\overline{WCF}(f_1) = \overline{WCF}(f_2)$ et $\underline{BCL}(f_1) = \underline{BCL}(f_2)$. Le meilleur cas de cohérence est nul car les intervalles $[\overline{WCF}(f_1), \underline{BCL}(f_1)]$ et $[\overline{WCF}(f_2), \underline{BCL}(f_2)]$ sont identiques. Le pire cas est :

$$\overline{WCCC}(\mathcal{F}_1, \mathcal{F}_2) = \overline{WCF}(f_1) - \underline{BCL}(f_2) = 170,82 - 0,682 = 170,138ms \tag{C.28}$$

Liste des publications

Conférences internationales avec comité de lecture

- Michaël Lauer, Jérôme Ermont, Frédéric Boniol and Claire Pagetti. Worst case temporal consistency in integrated modular avionics systems. In *Proceedings of the 13th IEEE International High Assurance Systems Engineering Symposium*, HASE'11, Boca Raton, Florida, November 2011.

- Michaël Lauer, Jérôme Ermont, Frédéric Boniol and Claire Pagetti. Latency and freshness analysis on IMA systems. In *Procedings of Emerging Technologies and Factory Automation*, ETFA'11, Toulouse, France. IEEE, september 2011.

- Michaël Lauer, Jérôme Ermont, Claire Pagetti and Frédéric Boniol. Analysing end-to-end functional delays on an IMA platform. In *Proceedings of the 4th international conference on Leveraging applications of formal methods, verification, and validation*, ISoLA'10, Heraklion, Crete, Greece, pages 243–257. Springer-Verlag, 2010.

- Michaël Lauer, Jérôme Ermont, Frédéric Boniol and Claire Pagetti. An interval-based method for embedded network analysis. In *Proceedings of the 3rd Junior Researcher Workshop on Real-Time Computing*, JRWRTC'09, Paris, France, pages 7–10. HAL-INRIA, 2009.

Conférence nationale avec comité de lecture

- Michaël Lauer, Jérôme Ermont, Frédéric Boniol and Claire Pagetti. Analyse de latence et de fraîcheur pire cas sur systèmes avioniques modulaires intégrées. In *Proceedings of 8ème Colloque Francophone sur la Modélisation des Sytèmes Réactifs*, MSR'11, Lille, France, November 2011.

Bibliographie

[1] Luca Aceto, Patricia Bouyer, Augusto Burgueño, and Kim Guldstrand Larsen. The power of reachability testing for timed automata. In *Proceedings of the 18th Conference on Foundations of Software Technology and Theoretical Computer Science*, FSTTCS'98, pages 245–256. Springer, December 1998.

[2] Muhammad Adnan, Jean-Luc Scharbarg, Jérôme Ermont, and Christian Fraboul. Model for worst-case delay analysis of an AFDX network using timed automata (short paper). In *Proceedings of the 15th IEEE International Conference on Emerging Technologies and Factory Automation*, ETFA'2010, Bilbao, Spain. IEEE, September 2010.

[3] Muhammad Adnan, Jean-Luc Scharbarg, Jérôme Ermont, and Christian Fraboul. An improved timed automata model for computing exact worst-case delays of afdx periodic flows. In *Proceedings of the 16th IEEE International Conference on Emerging Technologies Factory Automation*, ETFA'11, Toulouse, France, September 2011.

[4] Muhammad Adnan, Jean-Luc Scharbarg, and Christian Fraboul. Minimizing the search space for computing exact worst-case delays of AFDX periodic flows. In *Proceedings of the 6th IEEE International Symposium on Industrial Embedded Systems*, SIES'11, Vasteras, Sweden, pages 294 –301, June 2011.

[5] Ahmad Al Sheikh. *Resource allocation in hard real-time avionic systems Scheduling and routing problems*. PhD thesis, Institut National des Sciences Appliquées de Toulouse (INSA Toulouse), Toulouse, France, September 2011.

[6] Ahmad Al Sheikh, Olivier Brun, and Pierre-Emmanuel Hladik. Partition Scheduling on an IMA Platform with Strict Periodicity and Communication Delays. In *Proceedings of the 18th International Conference on Real-Time and Network Systems*, RTNS'10, Toulouse, France, pages 179–188, 2010.

[7] Rajeev Alur. *Techniques for automatic verification of real-time systems*. PhD thesis, Stanford University, CA, USA, August 1991.

[8] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126 :183–235, April 1994.

[9] Charles André. Syntax and Semantics of the Clock Constraint Specification Language (CCSL). Research Report RR-6925, INRIA, 2009.

[10] Charles André and Frédéric Mallet. Modèle de contraintes temporelles pour systèmes polychrones. *Journal Européen des Systèmes Automatisés*, 7-9(43) :725–739, June 2009.

[11] Charles André, Frédéric Mallet, and Robert De Simone. Modeling Time(s). In *Proceedings of the ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, volume LNCS 4735 of *MoDELS/UML'07*, pages 559–573. Springer, 2007.

[12] Aeronautical Radio Inc. ARINC 653. *Avionics Application Software Standard Interface*, 1997.

[13] Aeronautical Radio Inc. ARINC 664. *Aircraft Data Network Part 7 : "Avionics Full Duplex Switched Ethernet (AFDX) Network"*, 2005.

[14] Neil C. Audsley, Alan Burns, Mike F. Richardson, Ken Tindell, and Andy J. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8(5) :284 –292, September 1993.

[15] François Baccelli and Pierre Brémaud. *Elements of queueing theory : Palm-Martingale Ccalculus and Stochastic Recurrences*. Applications of mathematics. Springer, 2003.

[16] Henri Bauer. *Analyse pire cas de ux hétérogènes dans un réseau embarqué avion.* PhD thesis, Université de Toulouse, Toulouse, France, 2011.

[17] Henri Bauer, Jean-Luc Scharbarg, and Christian Fraboul. Applying and optimizing Trajectory approach for performance evaluation of AFDX avionics network. In *Proceedings of the 14th IEEE International Conference on Emerging Technologies and Factory Automation*, ETFA'09, Palma de Mallorca, Spain, pages 690–697. IEEE Press, 2009.

[18] Henri Bauer, Jean-Luc Scharbarg, and Christian Fraboul. Improving the worst-case delay analysis of an AFDX network using an optimized trajectory approach. *IEEE Transactions on Industrial Informatics*, 6(4) :521–533, November 2010.

[19] Razieh Behjati, Tao Yue, Shiva Nejati, Lionel Briand, and Bran Selic. An AADL-Based SysML profile for architecture level systems engineering : Approach, metamodels, and experiments. Technical Report 2011-03, Simula Research Laboratory, February 2011.

[20] Gerd Behrmann, Alexandre David, Kim G. Larsen, John Hakansson, Paul Petterson, Wang Yi, and Martijn Hendriks. UPPAAL 4.0. In *Proceedings of the 3rd International Conference on the Quantitative Evaluation of SysTems*, QEST'06, Riverside, CA, USA, pages 125–126. IEEE Computer Society, 2006.

[21] Johan Bengtsson and Wang Yi. Timed automata : Semantics, algorithms and tools. Research Report 3166, UNU/IIST, September 2004.

[22] Béatrice Bérard, Michel Bidoit, Alain Finkel, François Laroussinie, Antoine Petit, Laure Petrucci, and Philippe Schnoebelen. *Systems and Software Verification : Model-Checking Techniques and Tools.* Springer, 2001.

[23] Béatrice Bérard and L. Sierra. Comparing verification with hytech, kronos and uppaal on the railroad crossing example. Research Report LSV-00-2, Laboratoire Spécification et Vérification, ENS Cachan, France, 2000.

[24] Michel Berkelaar, Kjell Eikland, and Peter Notebaert. lp_solve 5.5, open source (mixed-integer) linear programming system. Software, May 2004. Available at <http ://lp-solve.sourceforge.net/5.5/>. Last accessed Dec, 18 2009.

[25] Ahmed Bouajjani, Stavros Tripakis, and Sergio Yovine. On-the-fly symbolic model checking for real-time systems. In *Proceedings of the 18th IEEE Real-Time Systems Symposium*, RTSS'97, San Francisco, CA, USA, pages 232–243. IEEE, December 1997.

[26] Anne Bouillard, Bertrand Cottenceau, Bruno Gaujal, Laurent Hardouin, Sebastien Lagrange, Mehdi Lhommeau, and Eric Thierry. COINC Library : A toolbox for Network Calculus. In *Proceedings of the 4th International Conference on Performance Evaluation Methodologies and Tools*, VALUETOOLS'09, Pisa, Italy, 2009.

[27] Anne Bouillard, Laurent Jouhet, and Eric Thierry. Tight performance bounds in the worst-case analysis of feed-forward networks. In *Proceedings of the 29th IEEE International Conference on Computer Communications*, INFOCOM'10, San Diego, CA, USA, pages 1316–1324, March 2010.

[28] Mokrane Bouzeghoub. A framework for analysis of data freshness. In *Proceedings of the 2004 International Workshop on Information Quality in Information Systems*, IQIS'04, Paris, France, pages 59–67. ACM, 2004.

[29] Marc Boyer. NC-Maude : A rewriting tool to play with network calculus. In *Proceedings of the 4th International Symposium on Leveraging Applications*, volume 6415 of *ISoLA'10, Heraklion, Crete, Greece*, pages 137–151. Springer-Verlag/Heidelberg, 2010.

[30] Marc Boyer and David Doose. Collaboration entre méthode d'ordonnancement et calcul réseau. In *Proceedings of Workshop on Approches Formelles des Systèmes Embarqués Communicants*, AFSEC'10, Pau, France, Pau, March 2010.

[31] Marc Boyer and Christian Fraboul. Tightening end to end delay upper bound for AFDX network calculus with rate latency FCFS servers using network calculus (regular paper). In *Proceedings of IEEE International Workshop on Factory Communication Systems*, WFCS'08, Dresden, Germany, pages 11–20. IEEE, May 2008.

[32] Matthias Brun, Jérôme Delatour, and Yvon Trinquet. Code generation from AADL to a real-time operating system : An experimentation feedback on the use of model transformation. In *Proceedings of the 13th IEEE International Conference on on Engineering of Complex Computer Systems*, ICECCS'08, Belfast, Northern Ireland, pages 257–262. IEEE Computer Society, 2008.

[33] Augusto Burgueño Arjona. *Vérification et synthèse de systèmes temporisés par des méthodes d'observation et d'analyse paramétrique (in english).* PhD thesis, Ecole Nationale Supérieure de l'Aéronautique et de l'Espace (SupAéro), Toulouse, France, June 1998.

[34] François Carcenac. *Un modèle d'abstraction pour la vérification des systèmes embarqués distribués : application à l'avionique.* PhD thesis, Ecole Nationale Supérieure de l'Aéronautique et de l'Espace (SupAéro), Toulouse, France, December 2005.

[35] François Carcenac and Frédéric Boniol. A formal framework for verifying distributed embedded systems based on abstraction methods. *International Journal on Software Tools for Technology Transfer (STTT)*, 8(6), 2006.

[36] Cheng-Shang Chang. *Performance Guarantees in Communication Networks.* Springer, 2000.

[37] Hussein Charara, Jean-Luc Scharbarg, Jérôme Ermont, and Christian Fraboul. Methods for bounding end-to-end delays on an AFDX network. In *Proceedings of the 18th Euromicro Conference on Real-Time Systems*, ECRTS'06, Dresden, Germany, pages 193–202. IEEE, July 2006.

[38] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking.* The MIT Press, 1999.

[39] Rene L. Cruz. A calculus for network delay, part 1 : Network elements in isolation. *IEEE Transactions on Information Theory*, 37(1) :114–131, 1991.

[40] Rene L. Cruz. A calculus for network delay, part 2 : Network analysis. *IEEE Transactions on Information Theory*, 37(1) :132–141, 1991.

[41] George B. Dantzig. Maximization of a linear function of variables subject to linear inequalities. *Activity Analysis of Production and Allocation*, pages 339–347, 1951.

[42] Vincent David, Jean Delcoigne, Evelyne Leret, Alain Ourghanlian, Philippe Hilsenkopf, and Philippe Paris. Safety properties ensured by the OASIS model for safety critical real-time systems. In *Proceedings of the 17th International Conference on Computer Safety, Reliability and Security*, SAFECOMP '98, pages 45–59, London, UK, 1998. Springer-Verlag.

[43] Julien Delange, Laurent Pautet, Alain Plantec, Mickael Kerboeuf, Frank Singhoff, and Fabrice Kordon. Validate, simulate, and implement ARINC 653 systems using the AADL. In *Proceedings of the ACM SIGAda annual international conference on Ada and related technologies*, SIGAda'09, Saint Petersburg, Florida, USA, pages 31–44. ACM, November 2009.

[44] Jérôme Ermont, Jean-Luc Scharbarg, and Christian Fraboul. Timed analysis of embedded networks using timed automata. In *Proceedings of the 7th IFAC International Conference On Fieldbuses and Networks in Industrial and Embedded Systems*, FeT'07, Toulouse, France, pages 255–260. LAAS, November 2007.

[45] Madeleine Faugère, Thimothée Bourbeau, Robert De Simone., and Sébastien Gérard. MARTE : Also an UML Profile for Modeling AADL Applications. In *Proceedings of the 12th IEEE International Conference on Engineering Complex Computer Systems*, ICECCS'07, Auckland, New Zealand, pages 359 –364, July 2007.

[46] Peter H. Feiler, David P. Gluch, and John J. Hudak. The architecture analysis & design language (AADL) : An introduction. Technical Report CMU/SEI-2006-TN-011, Software Engineering Institute, Carnegie Mellon University, February 2006.

[47] Peter H. Feiler and J. Hansson. Flow latency analysis with the architecture analysis and design language (AADL). Technical Report CMU/SEI-2007-TN-010, Software Engineering Institute, Carnegie Mellon University, December 2007.

[48] Fabrice Frances, Christian Fraboul, and Jérôme Grieu. Using network calculus to optimize the AFDX network. In *Proceedings of the 3rd European Congress on Embedded Real-Time Software*, ERTS'06, Toulouse, France. SIA/3AF/SEE, January 2006.

[49] Ritzberger Fritz. RunCC – A Java Runtime Compiler Compiler. `http://runcc.sourceforge.net/`.

[50] Hubert Garavel and Holger Hermanns. On combining functional verification and performance evaluation using CADP. In *Proceedings of the 11th International Symposium of Formal Methods Europe on Formal Methods*, FME'02, Copenhagen, Denmark, pages 410–429. Springer-Verlag, July 2002.

[51] Hubert Garavel and Damien Thivolle. Verification of GALS systems by combining synchronous languages and process calculi. In *Proceedings of the 16th International SPIN Workshop on Model Checking Software*, SPIN'09, Grenoble, France, pages 241–260. Springer-Verlag, June 2009.

[52] Jérome Grieu. *Analyse et évaluation de techniques de commutation Ethernet pour l'interconnexion des systèmes avioniques.* PhD thesis, Institut National Polytechnique de Toulouse, Toulouse, France, September 2004.

[53] Object Management Group. A UML profile for MARTE. Technical report, Object Management Group, Inc, 2007.

[54] Object Management Group. OMG systems modeling language. Technical report, Object Management Group, Inc, 2010.

[55] Gerard Holzmann. *The Spin model checker : primer and reference manual.* Addison-Wesley Professional, 2003.

[56] Aoste Inria Sophia-antipolis. TimeSquare. `http://timesquare.inria.fr`.

[57] ISO/IEC 14977 :1996(E) First edition. Information technology – Syntactic metalanguage – Extended BNF. Technical report, ISO/IEC, 1996.

[58] Praveen Jayachandran and Tarek Abdelzaher. Delay composition algebra : A reduction-based schedulability algebra for distributed real-time systems. In *Proceedings of the 29th IEEE Real-Time Systems Symposium*, RTSS'08, Barcelona, Spain, pages 259–269. IEEE Computer Society, 2008.

[59] Praveen Jayachandran and Tarek F. Abdelzaher. Transforming distributed acyclic systems into equivalent uniprocessors under preemptive and non-preemptive scheduling. In *Proceedings of the 20th Euromicro Conference on Real-Time Systems*, ECRTS'08, Prague, Czech Republic, pages 233–242. IEEE Computer Society, July 2008.

[60] Abhay Kumar Jha, Ming Xiong, and Krithi Ramamritham. Mutual consistency in real-time databases. In *Proceedings of the 27th IEEE International Real-Time Systems Symposium*, RTSS'06, Rio de Janeiro, Brazil, pages 335–343. IEEE Computer Society, December 2006.

[61] Gilles Kahn. The semantics of simple language for parallel programming. In *Proceedings of the IFIP Congress*, Stockholm, Sweden, pages 471–475. North-Holland, 1974.

[62] Jan Korst, Emile Aarts, Jan Lenstra, and Jaap Wessels. Periodic multiprocessor scheduling. In *Proceedings of the 3rd International Conference on Parallel architectures and languages Europe*, volume 1 of *PARLE'91, Eindhoven, the Netherlands*, pages 166–178. Springer-Verlag, June 1991. 10.1007/BFb0035103.

[63] Kai Lampka, Simon Perathoner, and Lothar Thiele. Analytic real-time analysis and timed automata : a hybrid method for analyzing embedded real-time systems. In *Proceedings of the 9th ACM/IEEE International Conference on Embedded Software*, EMSOFT'09, Grenoble, France, pages 107–116. ACM, 2009.

[64] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21 :558–565, July 1978.

[65] Leslie Lamport. Introduction to TLA. Technical report, Digital Equipment Corporation, 1994.

[66] Ailsa H. Land and Alison G. Doig. An automatic method of solving discrete programming problems. *Econometrica*, 28(3) :497–520, July 1960.

[67] Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a Nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1–2) :134–152, 1997.

[68] Michaël Lauer, Jérôme Ermont, Frédéric Boniol, and Claire Pagetti. An interval-based method for embedded network analysis. In *Proceedings of the 3rd Junior Researcher Workshop on Real-Time Computing*, JRWRTC'09, Paris, France, pages 7–10. HAL-INRIA, 2009.

[69] Michaël Lauer, Jérôme Ermont, Frédéric Boniol, and Claire Pagetti. Analyse de latence et fraîcheur pire cas sur systèmes avioniques modulaires intégrées. In *Proceedings of 8ème Colloque Francophone sur la Modélisation des Systèmes Réactifs*, MSR'11, Lille, France, November 2011.

[70] Michaël Lauer, Jérôme Ermont, Frédéric Boniol, and Claire Pagetti. Latency and freshness analysis on IMA systems (regular paper). In *Proccedings of Emerging Technologies and Factory Automation*, ETFA'11, Toulouse, France. IEEE, september 2011.

[71] Michaël Lauer, Jérôme Ermont, Frédéric Boniol, and Claire Pagetti. Worst case temporal consistency in integrated modular avionics systems. In *Proceedings of the 13th IEEE International High Assurance Systems Engineering Symposium*, HASE'11, Boca Raton, Florida, November 2011.

[72] Michaël Lauer, Jérôme Ermont, Claire Pagetti, and Frédéric Boniol. Analyzing end-to-end functional delays on an IMA platform. In *Proceedings of the 4th international conference on Leveraging applications of formal methods, verification, and validation*, ISoLA'10, Heraklion, Crete, Greece, pages 243–257. Springer-Verlag, 2010.

[73] Wolfhard Lawrenz. *Can System Engineering : From Theory to Practical Applications*. Springer, 1997.

[74] Jean-Yves Le Boudec and Patrick Thiran. *Network Calculus : a theory of deterministic queuing systems for the internet*. Springer, Berlin, Heidelberg, 2004.

[75] Edward A. Lee and Alberto Sangiovanni-Vincentelli. The tagged signal model - a preliminary version of a denotational framework for comparing models of computation. Technical Report UCB/ERL M96/33, University of California, Berkeley, CA, June 1996.

[76] Edward A. Lee and Alberto Sangiovanni-Vincentelli. A framework for comparing models of computation. *Transactions on Computer-Aides Design of Integrated Circuits And Systems*, 17(12) :1217–1229, 1998.

[77] John P. Lehoczky. Fixed priority scheduling of periodic task sets with arbitrary deadlines. In *Proceedings of the 11th IEEE Real-Time Systems Symposium*, RTSS'90, Lake Buena Vista, Florida, USA, pages 201–213, December 1990.

[78] Xiaoting Li, Jean-Luc Scharbarg, and Christian Fraboul. Improving end-to-end delay upper bounds on an AFDX network by integrating offsets in worst-case analysis. In *Proceedings of the 15th IEEE International Conference on Emerging Technologies and Factory Automation*, ETFA'10, Bilbao, Spain, September 2010.

[79] Xiaoting Li, Jean-Luc Scharbarg, and Christian Fraboul. Analysis of the pessimism of the trajectory approach for upper bounding end-to-end delay of sporadic flows sharing a switched ethernet network. In *Proceedings of the 19th International Conference on Real-Time and Network Systems*, RTNS'11, Nantes, France, pages 149–158, September 2011.

[80] Chung Lang Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20 :46–61, January 1973.

[81] Jukka Mäki-Turja and Mikael Nolin. Efficient implementation of tight response-times for tasks with offsets. *Real-Time Systems*, 40 :77–116, October 2008.

[82] Ricardo Santo Marques, Nicolas Navet, and Françoise Simonot-Lion. Configuration of in-vehicle embedded systems under real-time constraints. In *Proceedings of the 10th IEEE International Conference on Emerging Technologies and Factory Automation*, ETFA'05, Catania, Italy. IEEE, 2005.

[83] Steven Martin. *Maîtrise de la dimension temporelle de la qualité de service dans les réseaux*. PhD thesis, Université Paris XII, 2004.

[84] Steven Martin and Pascale Minet. Schedulability analysis of flows scheduled with fifo : application to the expedited forwarding class. In *Proceedings of the 20th International Conference on Parallel and Distributed Processing Symposium*, IPDPS'06, Rhodes Island, Greece. IEEE Computer Society, 2006.

[85] Steven Martin and Pascale Minet. Worst case end-to-end response times of flows scheduled with fp/fifo. In *Proceedings of the 5th International Conference on Networking, International Conference on Systems and International Conference on Mobile Communications and Learning Technologies*, ICN/ICONS/MCL'06, Mauritius. IEEE Computer Society, April 2006.

[86] Jörn Martin Migge and Alain Jean-Marie. Timing Analysis of Real-Time Scheduling Policies : A Trajectory Based Model. Technical Report RR-3561, INRIA, 1998.

[87] Jörn Martin Migge and Alain Jean-Marie. Real-Time Scheduling : Non-Preemption, Critical Sections and Round Robin. Technical Report RR-3678, INRIA, 1999.

[88] José Carlos Palencia and Michael González Harbour. Schedulability analysis for tasks with static and dynamic offsets. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*, RTSS'98, Madrid, Spain. IEEE Computer Society, December 1998.

[89] Nadège Pontisso. *Association cohérente de données dans les systèmes temps réel à base de composants - Application aux logiciels spatiaux*. PhD thesis, Institut National Polytechnique de Toulouse, December 2009.

[90] Nadège Pontisso, Philippe Quéinnec, and Gérard Padiou. Analysis of Distributed Multi-Periodic Systems to Achieve Consistent Data Matching. In *Proceedings of the 10th International Conference on NOuvelles TEchnologies de la REpartition*, NOTERE'10, Tozeur, Tunisia, pages 81–88. IEEE, 2010.

[91] Krithi Ramamritham, Sang H. Son, and Lisa Cingiser DiPippo. Real-Time Databases and Data Services. *Real-Time Systems*, 28 :179–215, 2004.

[92] Wolfgang Reisig. *Petri nets : an introduction*. Springer-Verlag, 1985.

[93] Jean-François Rolland. *Développement et validation d'architectures dynamiques*. PhD thesis, Université Toulouse III – Paul Sabatier, Toulouse, France, 2008.

[94] SAE. Annex X Behavior Annex (AS5506-X draft-2.13), August 2010.

[95] Laurent Sagaspe, Gérard Bel, Pierre Bieber, Frédéric Boniol, and Charles Castel. Safe allocation of avionics shared resources. In *Proceedings of the 9th IEEE International Symposium on High Assurance Systems Engineering*, HASE'05, Heidelberg, Germany, pages 25–33. IEEE Computer Society, October 2005.

[96] Luca Santinelli and Liliana Cucu-Grosjean. Towards Probabilistic Real-Time Calculus. In *Proceedings of the 3rd Workshop on Compositional Theory and Technology for Real-Time Embedded Systems*, CRTS'10, San Diego, CA, USA, 2010.

[97] Jens B. Schmitt, Frank A. Zdarsky, and Ivan Martinovic. Performance bounds in feedforward networks under blind multiplexing. Technical Report 349/06, University of Kaiserslautern, Germany, 2006.

[98] Ahmad Al Sheikh, Olivier Brun, Pierre-Emmanuel Hladik, and Balakrishna Prabhu. A best-response algorithm for multiprocessor periodic scheduling. In *Proceedings of the 23rd Euromicro Conference on Real-Time Systems*, ECRTS'11, Porto, Portugal, pages 228–237. IEEE Computer Society, July 2011.

[99] Frank Singhoff, Jérôme Legrand, Laurent Nana, and Lionel Marcé. Cheddar : a flexible real time scheduling framework. In *Proceedings of the ACM SIGAda annual international conference on Ada*, SIGAda'04, Atlanta, Georgia, USA. ACM Press, November 2004.

[100] Oleg Sokolsky and Alexander Chernoguzov. Performance analysis of AADL models using real-time calculus. In *Proceedings of the 16th Monterey Workshop*, volume 6028 of *Redmond, WA, USA*, pages 227–249. Springer-Verlag/Heidelberg, April 2010.

[101] Xiaohui Song and Jane W.S. Liu. Maintaining temporal consistency : Pessimistic vs. optimistic concurrency control. *IEEE transactions on knowledge and data engineering*, 7(5) :786–796, October 1995.

[102] Marco Spuri. Holistic Analysis for Deadline Scheduled Real-Time Distributed Systems. Research Report RR-2873, INRIA, 1996.

[103] Robert Tarjan. Depth-First Search and Linear Graph Algorithms. *SIAM Journal on Computing*, 1(2) :146–160, 1972.

[104] Lothar Thiele, Samarjit Chakraborty, and Martin Naedele. Real-time calculus for scheduling hard real-time systems. In *Proceedings of the International Symposium on Circuits and Systems*, ISCAS'00, Geneva, Switzerland, pages 101–104, May 2000.

[105] Ken Tindell, Alan Burns, and Andy J. Wellings. An extendible approach for analysing fixed priority hard real-time tasks. *Journal of Real-Time Systems*, 6 :133–151, March 1994.

[106] Ken Tindell, Alan Burns, and Andy J. Wellings. Calculating controller area network (CAN) message response times. *Control Engineering Practice*, 3(8) :1163 – 1169, 1995.

[107] Ken Tindell and John Clark. Holistic schedulability analysis for distributed hard real-time systems. *Microprocessing and Microprogramming*, 40 :117–134, April 1994.

[108] Ernesto Wandeler and Lothar Thiele. Optimal TDMA time slot and cycle length allocation for hard real-time systems. In *Proceedings of the 11th Asia and South Pacific Design Automation Conference*, ASP-DAC'06, Yokohama, Japan, pages 479–484. IEEE Press, January 2006.

[109] Ernesto Wandeler and Lothar Thiele. Real-Time Calculus (RTC) Toolbox. `http://www.mpa.ethz.ch/Rtctoolbox`, 2006.

[110] Ling Yin, F. Mallet, and Jing Liu. Verification of MARTE/CCSL Time Requirements in Promela/SPIN. In *Proceedings of the 16th IEEE International Conference on Engineering of Complex Computer Systems*, ICECCS'11, Las Vegas, NV, USA, pages 65–74, April 2011.

[111] Sergio Yovine. KRONOS : A verification tool for real-time systems. *STTT*, 1(1-2) :123–133, 1997.

Résumé

Dans le domaine de l'aéronautique, les systèmes embarqués ont fait leur apparition durant les années 60, lorsque les équipements analogiques ont commencé à être remplacés par leurs équivalents numériques. Dès lors, l'engouement suscité par les progrès de l'informatique fut tel que de plus en plus de fonctionnalités ont été numérisées. L'accroissement permanent de la complexité des systèmes a conduit à la définition d'une architecture appelée *Avionique Modulaire Intégrée* (IMA pour *Integrated Modular Avionics*). Cette architecture se distingue des architectures antérieures, car elle est fondée sur des standards (ARINC 653 et ARINC 664 partie 7) permettant le partage des ressources de calcul et de communication entre les différentes fonctions avioniques. Ce type d'architecture est appliqué aussi bien dans le domaine civil avec le Boeing B777 et l'Airbus A380, que dans le domaine militaire avec le Rafale ou encore l'A400M. Pour des raisons de sûreté, le comportement temporel d'un système s'appuyant sur une architecture IMA doit être prévisible. Ce besoin se traduit par un ensemble d'*exigences temps réel* que doit satisfaire le système. Le problème exploré dans cette thèse concerne la vérification d'exigences temps réel dans les systèmes IMA. Ces exigences s'articulent autour de *chaînes fonctionnelles*, qui sont des séquences de fonctions. Une exigence spécifie alors une borne acceptable (minimale ou maximale) pour une propriété temporelle d'une ou plusieurs chaînes fonctionnelles. Nous avons identifié trois catégories d'exigences temps réel, que nous considérons pertinentes vis-à-vis des systèmes étudiés. Il s'agit des exigences de *latence*, de *fraîcheur* et de *cohérence*. Nous proposons une modélisation des systèmes IMA, et des exigences qu'ils doivent satisfaire, dans le formalisme du *tagged signal model*. Nous montrons alors comment, à partir de ce modèle, nous pouvons générer pour chaque exigence un programme linéaire mixte, c'est-à-dire contenant à la fois des variables entières et réelles, dont la solution optimale permet de vérifier la satisfaction de l'exigence.

Mots clefs : systèmes embarqués, temps réel, modélisation, vérification

Abstract

Embedded systems appeared in aeronautics during the 60's, when the process of replacing analog devices by their digital counterpart started. From that time, the broad thrust of computer science advances make it possible to digitize more and more avionics functionalities. The continual increase of the complexity of these systems led to the definition of a new architecture called *Integrated Modular Avionics* (IMA). This architecture stands apart from previous architecture because it is based on standards (ARINC 653 and ARINC 664 part 7) which allow the sharing of computation and communication resources among avionics functions. This architecture is implemented in civil aircrafts, with Boeing B777 and Airbus A380, and in military aircrafts, with Rafale or A400M. For safety reason, the temporal behavior of such a system must be predictable, which is expressed with a set real-time requirements. A real-time requirement specifies an upper or lower bound of a temporal property of one or several functional chains. A functional chain is a sequence of functions. In this thesis, we explore the verification of real-time requirements in IMA systems. We have identified three real-time requirements relevant to our problem : *latency*, *freshness* and *consistency*. We propose a model of IMA systems, and the requirements they must meet, based on the *tagged signal model*. Then we derive from this model, for each requirement, a mixed integer linear program whose optimal solution allows us to verify the requirement.

Keywords : embedded system, real-time, modeling, verification